

Z-Wave Humidity-Temperature Sensor Project

The demo programs that come with the Z-Wave 7.0 DevKit don't involve much work. You change a constant for the radio frequency, compile, flash the board, and press a few buttons and watch a few LEDs. The documentation recommends as a next step extending one of the demo programs for your application. But the project setup is complicated, the interaction with the Z-Wave Application Framework (ZAF) opaque, and how to make changes unclear.

We set out to write a demo program that would fill in these gaps. For one, the devkit board includes some hardware that isn't used in any of the demos, so our program will use it: a humidity-temperature sensor and the LCD. For another, we'll start with a blank project to learn what's needed for libraries and hardware drivers, how code is organized within Simplicity Studio, and how the build process works. For a third, we'll add functionality and commands within the framework to see what it does and how its components are put together. Our demo will allow you to read the sensor via Z-Wave or to start a data logging run that will periodically send readings to the controller; in both cases we'll also see the readings on the LCD.

The program will be built in several steps. First we need to set up the equivalent of a "Hello World" program within the application framework, the simplest program that will compile. Second we'll figure out how to work with the hardware, first the buttons and LEDs, then the humidity-temperature sensor, then the LCD. These steps will involve working with libraries and code in the SDK. Third we'll add Z-Wave support, first inclusion and exclusion, then getting a correct Node Information Frame (NIF) and adding CC_VERSION, next implementing CC_SENSOR_MULTILEVEL to read the sensor and CC_CONFIGURATION for starting a logging run, and finishing by setting up security (just S0). Fourth we'll look at the differences between our demo and the full programs provided with the DevKit.

This last point is important. Undoubtedly there are FAEs at Silicon Labs that will shake their heads at what we're documenting here: it may not follow best practices, it doesn't follow their coding standards, and it doesn't try to be Z-Wave compliant. The demo is meant to be a starting point to unravel a rather large knot. We hope the result provides the background one needs to develop a good production program.

Here is a [tarball](#) with the entire project; you'll also find links to separate tarballs for each step at the start of their section. If it's easier to read this text in one document, you'll find a [PDF here](#).

Step 1: [Setup and Framework](#)

Step 2a: [Hardware - Buttons and LEDs](#)

Step 2b: [Hardware - Humidity/Temperature Sensor](#)

Step 2c: [Hardware - LCD](#)

Step 3a: [ZWave - Inclusion and Exclusion](#)

Step 3b: [ZWave - NIF](#)

Step 3c: [ZWave - CC_SENSOR_MULTILEVEL \(reading\)](#)

Step 3d: [ZWave - CC_CONFIGURATION \(logging\)](#)

Step 3e: [ZWave - Security](#)

Step 4: [Application Framework](#)

The Silicon Labs specs we used in developing the program include

[UG381 ZGM130S Zen Gecko Wireless Starter Kit User's Guide](#)

devkit hardware description

[INS14280 Z-Wave 700 Getting Started for End Devices](#)

devkit bring-up, first demo program

[INS14278 How To Use Certified Apps in Z-Wave 700](#)

demo program description

[INS14259 Z-Wave Plus V2 Application Framework SDK7](#)

framework description and how to use

[INS14281 Z-Wave 700 Getting Started for Controller Devices](#)

Z/IP gateway program and GUI

[SDS13781 Z-Wave Application Command Class Specification](#)

most command classes

[SDS13812 Multilevel Sensor Command Class](#)

list of assigned Multilevel Sensor types and scales

DSH14299 ZGM130S Z-Wave 700 DiP Module Data Sheet

part of the Simplicity Studio SDK documentation

[Si7021-A20 I2C Humidity and Temperature Sensor Data Sheet](#)

chip specification, also how to read values

[LCD Module LS013B7DH03 Device Specification](#)

signal timing

We're using the Z-Wave 700 DevKit and Simplicity Studio 4.2, running under Linux. The SDK was version 7.12.1. We will need two files from the Gecko SDK, version 2.4. We're using the Python GUI for the Z-Wave gateway software to communicate with the device, and our [Zpiffer program](#) to monitor network traffic.

One last note. Much of the text will describe the evolution of the source code as we build on previous steps. There will be links to code snippets showing the most important changes or functionality, but if anything is unclear, try looking at a diff of the file to the previous step or the original source in the SDK. Let us know if something still doesn't make sense so we can fix the text. As always, comments about the project or write-up are welcome at [by sending us an e-mail](#).

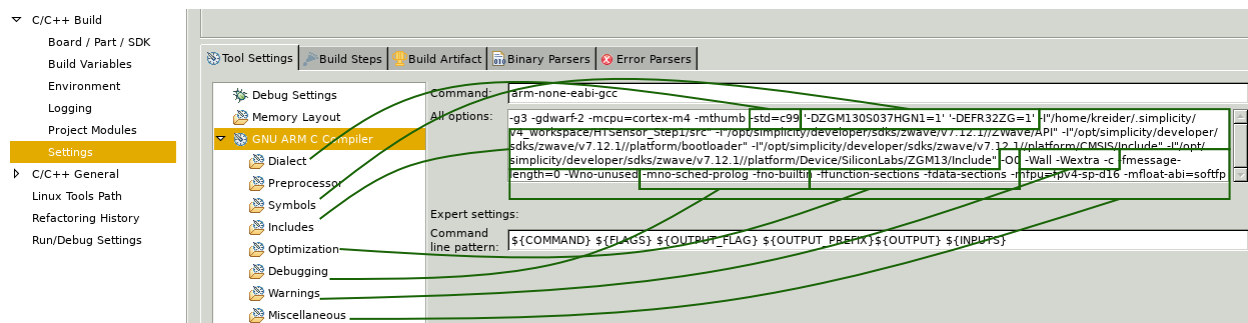
Step 1: Setup and Framework

The code for this part of the demo is found [here](#).

Build Process

A Java package within Simplicity Studio generates a makefile and directory hierarchy based on a project's settings, accessible by right clicking on a project in the Explorer window on the left and choosing Properties. The builder creates a directory named after the build configuration ('GNU ARM v7.2.1 - Debug' for example) and creates sub-directories for each directory in the project. It puts a subdir.mk makefile in each, which when run will create a header dependency list (*.d) and object file (*.o). The final binaries are put in the build directory.

The project settings are stored in the file .cproject. You can see how the commands in the makefile are built up from entries in the C/C++ Build Settings part of the Properties dialog (either by right-clicking on the project name in the Explorer, or under the Project menu entry). The long command line for the compiler/assembler/linker contains the Includes files (-I prepended to each) and Symbols (-D), with checkboxes in other entries adding flags, as well as the 'Other flags' list found under Miscellaneous. Compiling is done in the sub-directories by the subdir.mk files; the top makefile links the binary in AXF format and converts it to BIN, HEX, and S37 formats. Each source file has its own make target, although the commands are the same except for the file names.



Build command found in Project Properties. Arcs indicate source of command line options within dialog.

We prefer to link to source files in the SDK rather than copying them into the project. If done within Simplicity Studio the links do not appear in the file system. They'll appear in the Project Explorer window with an arrow in the file icon, and will be listed in the Linked Resources/Linked Resources entry of the Project Settings window. When creating the Makefile the Java package will include a target for each source file with the link path expanded. Header files, however, cannot be treated this way. They must either be found in a directory in the include path, or you must symlink them in the file system (that is, not link them within Simplicity Studio).

You'll find abbreviations for some directories under Resource/Linked Resources/PathVariables, of which STUDIO_SDK_LOC is useful, and two more under C/C++ Build/Build Variables, which provides StudioSdkPath to use within the build dialog.

Resource

Linked Resources

Resource Filters

Builders

C/C++ Build

Board / Part / SDK

Build Variables

Environment

Logging

Project Modules

Settings

C/C++ General

Linux Tools Path

Refactoring History

Path Variables

Linked Resources

Path variables specify locations in the file system, including other path variables with the syntax "\${VAR}". The locations of linked resources may be specified relative to these path variables.

Defined path variables for resource 'HTSensor_Step1':

Name	Value
ECLIPSE_HOME	/opt/simplicity/
PARENT_LOC	./simplicity/v4_workspace
PROJECT_LOC	./simplicity/v4_workspace/HTSensor_Step1
STUDIO_APACK_COMMANDER_LOC	/opt/simplicity/developer/adaptor_packs/commander/commander
STUDIO_APACK_LOC	
STUDIO_SDK_LOC	/opt/simplicity/developer/sdks/zwave/v7.12.1
STUDIO_TOOLCHAIN_LOC	/opt/simplicity/developer/toolchains/gnu_arm/7.2_2017q4
WORKSPACE_LOC	./simplicity/v4_workspace

C/C++ Build

Board / Part / SDK

Build Variables

Name	Type	Value
StudioSdkPath	Directory	\${StudioSdkPathFromID:com.silabs.sdk.stack.zwave:7.12.1.1_1539648630}
StudioToolchainPath	Directory	\${StudioToolchainPathFromID:com.silabs.ss.tool.ide.arm.toolchain.gnu.cdt:7.2.1.20170904}

Project Properties defining abbreviations. Use STUDIO_SDK_LOC except in the build settings, where the variable is StudioSdkPath.

You can delete the build directory ('GNU ARM v7.2.1 - Debug') at any time and the Build command within Simplicity Studio will recreate it. Once it exists, if the compiler is on your path, you can run make from the command line. It's found at simplicity/developer/toolchains/gnu_arm/7.2_2017q4/bin/. It might be possible to recreate the makefile from the .cproject information, but it seems easiest to let the Java program do the work.

SDK Contents

We'll see as we go along that the files and libraries we need are scattered throughout the SDK directory.

To keep paths short, we'll use SDK for <simplicity install dir>/developer/sdks/zwave/v7.12.1. For SDK you should substitute STUDIO_SDK_LOC everywhere within Simplicity Studio, except in the build settings when you would use StudioSdkPath.

directory	contents
\${SDK}/ZWave/lib	slave and controller libraries
\${SDK}/ZWave/API	header files
\${SDK}/ZAF/ApplicationUtilities	application framework
\${SDK}/ZAF/CommandClasses	some ZWave commands
\${SDK}/util/third-party/freertos/Source	FreeRTOS source
\${SDK}/ThirdParty/Freertos/include	FreeRTOS headers
\${SDK}/Processor/gecko	devkit configuration, including pins, clocks, UART
\${SDK}/Components	modules for timers, assertions, event groups, queues
\${SDK}/hardware/kit	display hardware and I2C bus
	devkit board is EFR32FG13_BRD4255A
\${SDK}/platform/emdrv	micro-controller drivers
\${SDK}/platform/emlib	peripherals library
\${SDK}/platform/Device/SiliconLabs/ZGM13	devkit board hardware access
\${SDK}/platform/bootloader	boot code
\${SDK}/platform/CMSIS	another hardware abstraction layer
\${SDK}/Apps	ZWave demo programs

ZWave Application Framework

The ZWave Application Framework is built atop FreeRTOS, a simple operating system. The OS provides tasks (processes/threads), events for communication, and queues for storing them. It offers semaphores/mutexes to synchronize access to queues and other resources. Timers can trigger one-off callbacks at a point in the future, or fire periodically. At a low level the OS has interrupt support and memory allocation.

We can group the framework's files into the following categories:

1. **Framework** - common defines and access to global variables.
2. **Hardware** - devkit components including LEDs and indicator, and buttons and sliders; on-chip blocks like I/O pins, an ADC, and UARTs. The key file is board.h.
3. **ZWave Network** - learn mode and SmartStart. The key file is ZAF_network_learn.h to monitor progress, although most of the process happens automatically within the framework.
4. **ZWave Communication** - outgoing packets (with separate buffers for requests and responses) and incoming frame router (per frame type or command, either explicitly for application commands or indirectly via the CommandPublisher), endpoint management, lifeline support, multicasting, security. The key files are ZW_TransportSecProtocol.h and ZW_TransportEndpoint.h, and ZAF_CmdPublisher.h to route frames to command classes.
5. **Events** - user event ("job") queues. The key file is ev_man.h.
6. **Software Timer** - interface to timer, EM4 hibernation management. The key file is AppTimer.h.
7. **Non-Volatile Memory** - access to persistent storage of objects. The key file is ZAF_nvm3_app.h.
8. **Power Management** - chip power mode. The only file is ZAF_PM_Wrapper.h.
9. **Command Classes** - implementation of several common ZWave commands.

The framework essentially sets up an event loop, with three handlers for incoming frames, ZWave events from

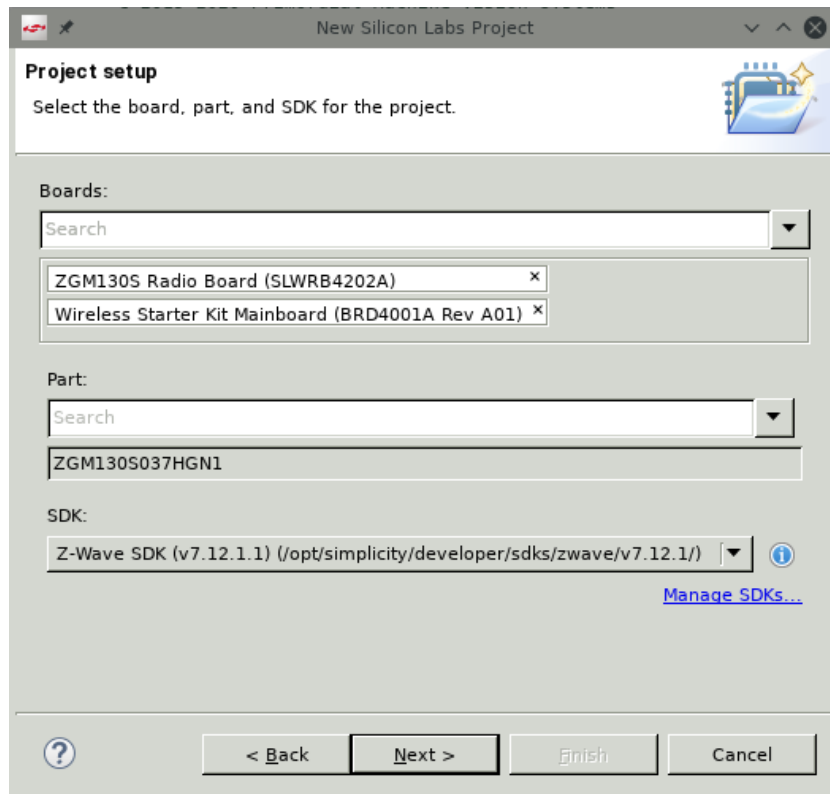
frames ZAF has handled, and hardware events. Dealing with incoming packets is a bit confusing. One handler, registered with the framework, routes based on the type of packet: single- or multi-cast, node update, or security event. A second, implementing a stub in the transport protocol, gets the frame's contents unpacked into a per-command class structure (which are grouped in a giant union) and must send it to the appropriate command class handler. A third router, the CommandPublisher, can be called within the first, single-cast, handler; it will do this unpacking and itself call the command class. Outgoing packets are easier, as the command classes put them in the appropriate REQUEST or RESPONSE transmit buffer.

Creating A Project

We want to start with a blank slate and do the minimum necessary to get a project to build. To do this we created a single source file with a single function definition, then iteratively changed project and build settings or linked files or added to the source, fixing each compile problem that arose. Our project directory structure will be a bit different than the demo programs; see the table for the layout. You may notice Simplicity Studio making several more entries under the project. The build directory 'GNU ARM v7.2.1 - Debug' exists on disk. 'Includes' and 'Binaries' do not. You will see them only within Simplicity Studio. The first are the header files from the Project Properties list, the second has links to the object files.

directory	custom files (on disk)	linked files
hw/	LCD drivers	other hardware files
src/	application, ZAF headers, I2C header	
ZAF_AppUtil/	modified framework files	ZAF source files
ZAF_CC/	ZWave command classes	default command classes
GNU ARM v7.2.1 - Debug/	none	auto-generated directory

To begin, choose Project -> New -> Silicon Labs MCU Project from the menu. If the DevKit board is plugged in the boards, part, and SDK will be filled in correctly. Pick Empty Project then enter a name.



First dialog to create a new project. Boards, Part, and SDK are automatically detected by Simplicity Studio. Follow the Next chain from here.

Open the Project Properties dialog and verify Simplicity Studio has created the four Linked Resources/Path Variables: STUDIO_SDK_LOC, STUDIO_TOOLCHAIN_LOC, STUDIO_APACK_COMMANDER_LOC, STUDIO_APACK_LOC.

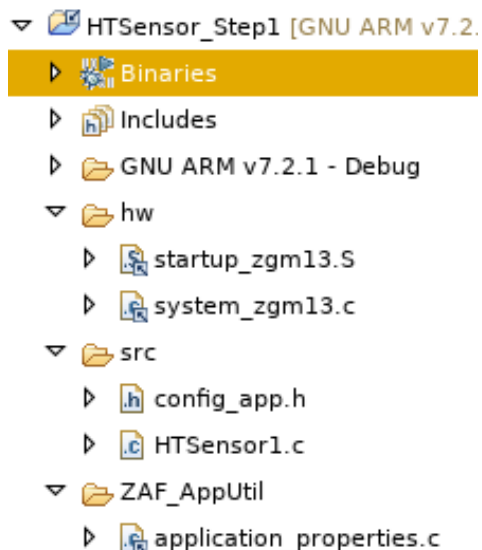
Create the file src/HTSensor1.c and put one function, [ApplicationInit\(\)](#), in it.

To get this to compile requires:

1. Adding `${SDK}/ZWave/API` to the C/C++ Build/Settings Include list.
2. Adding the path `${SDK}/ZWave/lib` and the library `ZWaveSlave` to Linker/Libraries.
3. Setting the linker script under Properties/Memory Layout to `${SDK}/ZWave/linkerscripts/zgm13-zw700.ld`.
4. Creating a folder `hw/` under the project.
5. Linking the file `${SDK}/platform/Device/SiliconLabs/ZGM13/Source/system_zgm13.c` under `hw/`. This is code for the devkit board clocks.
6. Linking `${SDK}/platform/Device/SiliconLabs/ZGM13/Source/GCC/startup_zgm13.S` under `hw/`. This is code called when the devkit board boots, for example registering interrupt handlers.
7. Adding `${SDK}/platform/Device/SiliconLabs/ZGM13/Include` to the C/C++ Include list.

8. Adding 'ZGM130S037HGN1=1' to the C/C++ Symbols list. This includes the right header file for the board.
9. Adding `${SDK}/platform/CMSIS/Include` to the C/C++ Include list.
10. Adding the path `${SDK}/SubTree/rail-import/platform/radio/rail_lib/autogen/librail_release` and the library `rail_efr32xg13_gcc_release` to Linker/Libraries. This is a library for testing and adjusting the radio when bringing up a new design.
11. Linking the file `${SDK}/ZAF/ApplicationUtilities/application_properties.c` under ZAF_AppUtil/. The bootloader uses this code to determine the application version.
12. Adding 'EMR32ZG=1' to the C/C++ Symbols list. If not defined `application_properties.c` will be an empty file.
13. Creating the file [src/config_app.h](#). The minimum contents define the symbols APP_VERSION, APP_REVISION, APP_PATH, APP_MANUFACTURER_ID, APP_PRODUCT_ID, and an enum _PRODUCT_TYPE_ID_ENUM; all are needed by the boot loader for version tracking. The values are placeholders and will not pass ZWave certification (especially the manufacturer ID).
14. Adding `${SDK}/platform/bootloader` to the C/C++ Include list.
15. Adding `${workspace_loc}/${ProjName}/src` to the C/C++ Include list, where ProjName is whatever you used when creating the project.

Use `${STUDIO_SDK_LOC}` instead of our abbreviation `${SDK}`, except when changing values in the Project Properties dialog; then, use `${StudioSdkPath}`. All file linking is done within Simplicity Studio by right clicking in the Project Explorer and choosing New -> File, then checking the Link box under the Advanced button and filling in the path.



Project contents. Note linked files (with arrows on icons) exist only in Simplicity Studio, not on disk.

Or in table form (since the listing will soon be too long for a screenshot)

hw/	src/	ZAF_AppUtil/	ZAF_CC/
	config_app.h HTSensor1.c		
<i>linked within Simplicity Studio</i>			
startup_zgm13.S system_zgm13.c		application_properties.c	

Stepping back, we've had to add boot code (ZGM13), libraries for Zwave and radio testing, and some application framework startup code to our simple "application". We added four directories to the include list, defined two symbols to enable hardware functionality, and linked to three source files that will be compiled along with our application. We also had to define a minimal set of version and product information for the framework.

The program will now compile. Object files are put in the build directory, and the Binaries entry in the project will have a .hex file that can be flashed to the devkit, as described in INS14280 Section 6. Nothing will happen, yet. Let's add in hardware support, starting with the [buttons and LEDs](#).

Step 2a: Hardware - Buttons and LEDs

The code for this step is found [here](#).

You must implement two functions for the framework, the first for initializing the hardware and application, the second the main task. A FreeRTOS task never returns, so the second function contains an infinite loop. The framework generates events for the system timer, hardware functions, or incoming ZWave traffic. These events are gathered in a queue that has been wrapped to handle event notifications (QueueNotify) and routing (EventDistributor); these wrappers are found in the Components directory in the SDK. The first thing our task must do is to create the event queue, wrap it, and register the result with the framework.

The demo programs define two sets of application events, putting one enumeration EVENT_APP in the file events.h and the other enumeration EApplicationEvent in the source code. The first uses a local "job" queue created for events outside the framework, and signals state changes during learning, non-volatile memory access, power and battery management, and within the application. We won't need to use these, and omit the file. The second is a group of four ZAF events, from the timer, incoming frames, ZWave state, and hardware and application events. These we do need. Each event needs a handler. ZAF provides one for the timer, we'll define one for the hardware, and leave a stub for the other two for now.

The expansion board has four buttons and four LEDs. Two of each tie to components on the main board. The functionality we'll eventually implement is

Button 1	ZWave inclusion/exclusion
Button 2	toggle data logging
Button 3	read temperature
Button 4	read humidity

We'll flicker LED2 while we're logging sensor readings, and turn LED4 on when a single read is in progress. LED1 is the indicator signalling that the node is in learn mode or is included. For logging, the number of samples to take and the time between them will be fixed for now, until we add a ZWave configuration command to set them. For Step 2a we'll use a timer to simulate the sensor read while driving LED4 correctly, and Button 2 will just toggle the logging mode (ie. turn LED2 on or off).

Implementation

There are two header files in the SDK that define button and LED operations. Only one, `board.h`, must be included in our application. The other will be found by passing a `-D` define to make, which is done by adding a symbol to the C/C++ Build items in the project's Properties. Including `board.h` requires adding many directories to the include path and linking files to the project:

1. Add `${SDK}/ZAF/ApplicationUtilities` to the C/C++ Include list.
2. Add `${SDK}/platform/emlib/inc` to the C/C++ Include list.
3. Add `'RADIO_BOARD_ZGM130S = 1'` to the C/C++ Symbols list.
4. Add `'EXT_BOARD_8029A = 1'` to the C/C++ Symbols list.
5. Link `${SDK}/ZAF/ApplicationUtilities/board.c` under `ZAF_AppUtil/`.
6. Add `${SDK}/platform/emdrv/gpiointerrupt/inc` to the C/C++ Include list.
7. Link `${SDK}/platform/emdrv/gpiointerrupt/src/gpiointerrupt.c` under `hw/`.
8. Link `${SDK}/ZAF/ApplicationUtilities/board_indicator.c` under `ZAF_AppUtil/`.
9. Link `${SDK}/ZAF/ApplicationUtilities/AppTimer.c` under `ZAF_AppUtil/`.
10. Link `${SDK}/platform/emlib/src/em_ltimer.c` under `hw/`.
11. Add `${SDK}/Components/SwTimer` to the C/C++ Include list.
12. Add `${SDK}/Components/DebugPrint` to the C/C++ Include list.
13. Link `${SDK}/ZAF/ApplicationUtilities/ZAF_uart_utils.c` under `ZAF_AppUtil/`.
14. Add `${SDK}/Components/Assert` to the C/C++ Include list.
15. Link `${SDK}/ZAF/ApplicationUtilities/EventHandling/zaf_event_helper.c` under `ZAF_AppUtil/`.
16. Add `${SDK}/Components/EventDistributor` to the C/C++ Include list.
17. Add `${SDK}/Components/QueueNotifying` to the C/C++ Include list.
18. Add `${SDK}/ZAF/ApplicationUtilities/EventHandling` to the C/C++ Include list.
19. Add `${SDK}/ThirdParty/Freertos/include` to the C/C++ Include list.
20. Add `${SDK}/ThirdParty/Freertos/include/gecko` to the C/C++ Include list.
21. Add `${SDK}/util/third_party/freertos/Source/portable/GCC/ARM_CM4F` to the C/C++ Include list.
22. Add `${SDK}/Components/Utils` to the C/C++ Include list.
23. Add `${SDK}/Components/NodeMask` to the C/C++ Include list.
24. Add `${SDK}/ZAF/ApplicationUtilities/PowerManagement` to the C/C++ Include list.
25. Modify `src/config_app.h` to include defines for `GENERIC_TYPE`, `SPECIFIC_TYPE`,

DEVICE_OPTIONS_MASK, MAX_TXPWR, and MEASURED_0DBM_TXPWR. The latter two will eventually end up in src/config_rf.h in Step 3.

Items #1-11 are needed to drive the devkit boards, with #5-7 for buttons, #8 for LEDs, and #9-11 for timers. Items #15-21 give us events and handlers for the framework. The others are support functions. In the application we use the DPRINT debug macro (#12) to write messages to the UART, which we can monitor in Simplicity Studio on the device console. This requires including the ZAF_uart_utils.h header file in our source file (#13). The SizeOf.h header (in #22) provides a macro sizeof_array which we use as a shortcut. AppTimer.h (from #9) provides a timer we can set and run within the application. The other header files we must include define functions and constants for the framework.

The application contains just a few functions. ApplicationInit() sets up the program and hardware, and ApplicationTask() finishes set up and contains the main event loop. We have one event handler, handle_event(), that reacts to button presses, and a callback simulate_ht() that runs when the timer that's pretending to be a sensor read finishes.

In the [ApplicationInit\(\)](#) function we do any set-up needed before starting the FreeRTOS application task. This includes the timer (call to the framework function AppTimerInit()), initializing framework variables (for the radio frequency, basic NIF information, and security), the devkit boards (framework function Board_Init()), and UART (framework function ZAF_UART0_enable()) once we're using the DebugPrint module. The function finishes by registering the task function, which allows FreeRTOS to start it.

The [ApplicationTask\(\)](#) function continues set-up before entering the event loop. We tell the timer about the task with AppTimerSetReceiverTask(), and finish framework setup by calling ZAF_Init(). We create a small event queue with the FreeRTOS function xQueueCreateStatic(), wrap it with the event notifier by calling QueueNotifyingInit() and ZAF_EventHelperInit(), and then set up the event router with EventDistributorConfig(). We finish hardware setup for the buttons with Board_EnableButton() and Board_SetLed(), and create a software timer to simulate sensor reads with AppTimerRegister(). Finally we go into the infinite loop, using EventDistributorDistribute() to direct events to the appropriate handler.

[handle_event\(\)](#) pulls events from the queue with xQueueReceive. It compares them to BTN_EVENT_SHORT_PRESS(BOARD_BUTTON_PB[1-4]) to determine which button has been pressed, and reacts appropriately. Use Board_SetLed(BOARD_LED[1-4], LED_[OFF|ON]) to change a LED, and TimerStart(&timer, time_in_ms) to create a small delay before using [simulate_ht\(\)](#) to turn off LED4.

The demo programs include many global variables. We've tried to reduce them as much as possible, only keeping those that aren't passed between framework functions or which must be statically allocated. The framework, for example, must always be able to see the NIF structure, radio configuration, and security keys, as well as an overall ZWave configuration. The event queue is needed in the handlers. Our application must have the timer always available, a buffer to hold the debug log, and a counter for the simulated log.

The call to ZAF_Init() pulls in still more files and include directories.

1. Add \${SDK}/ZAF/ApplicationUtilities/TrueStatusEngine to the C/C++ Include list.
2. Add \${SDK}/ZAF/CommandClasses/Common to the C/C++ Include list.

3. Link \${SDK}/ZAF/ApplicationUtilities/ZW_TransportSecProtocol.c under ZAF_AppUtil/.
4. Link \${SDK}/ZAF/ApplicationUtilities/ZW_TransportMulticast.c under ZAF_AppUtil/.
5. Link \${SDK}/ZAF/ApplicationUtilities/ZAF_command_class_utils.c under ZAF_AppUtil/.

Since we're not hooking up ZWave yet, we'll make local copies of some of the framework files and remove functionality so they compile. In Step 3 we'll restore the edits and talk more about what's going on. From \${SDK}/ZAF/ApplicationUtilities

1. Copy ZAF_TSE.c into ZAF_AppUtil/, remove the include of association_plus.h and comment out the call to handleAssociationGetnodeList(). This is so we don't have to pull in association groups.
2. Copy ZW_TransportEndpoint.c into ZAF_AppUtil/, remove the inclusion of multichannel.h, and comment out the call to CmdClassMultiChannelEncapsulate(). Also add a stub in our application program for Transport_ApplicationCommandHandlerEx(), which requires including the header files ZW_classcmd.h and ZAF_types.h (which are already on the include path).

The project now contains 17 files between the hw/, src/, and ZAF_AppUtil/ directories, although most are links to the SDK.

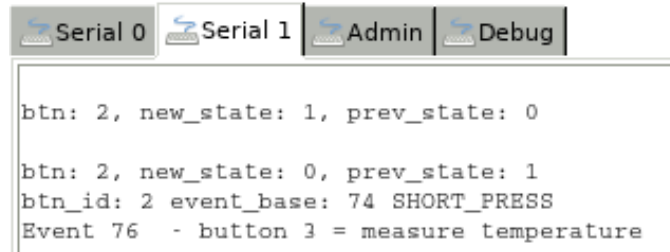
hw/	src/	ZAF_AppUtil/	ZAF_CC/
	config_app.h HTSensor1.c	ZAF_TSE.c ZW_TransportEndpoint.c	
<i>linked within Simplicity Studio</i>			
em_letimer.c gpinterrupt.c startup_zgm13.S system_zgm13.c		application_properties.c AppTimer.c board_indicator.c board.c ZAF_command_class_utils.c zaf_event_helper.c ZAF_uart_utils.c ZW_TransportMulticast.c ZW_TransportSecProtocol.c	

In our application we find six functions.

function	role	description
handle_event	button press	set LEDs, simulate sensor actions
handle_dummy	placeholder	empty ZWave command handler
simulate_ht	placeholder	turn off LED showing logging
ApplicationInit	ZAF	pre-task system setup
ApplicationTask	ZAF	post-task setup and event loop
Transport_ApplicationCommandHandlerEx	ZAF	empty ZWave command handler

The project compiles, the LEDs show button functionality is working, and you can monitor events in the debug log console (by right clicking on the J-Link adapter, launching a console, picking the Serial 1 tab, and typing a

return there to start the link).



```
Serial 1 | Admin | Debug  
btn: 2, new_state: 1, prev_state: 0  
btn: 2, new_state: 0, prev_state: 1  
btn_id: 2 event_base: 74 SHORT_PRESS  
Event 76 - button 3 = measure temperature
```

Console trace of button 3 press. The framework generates lines starting with 'btn'.

Next we'll access the [sensor](#).

Step 2b: Hardware - HT Sensor

The code for this step is found [here](#).

Just an aside, to copy a project in Simplicity Studio right-click in the Project Explorer and choose Import -> MCU Project. Type the existing project name in the dialog, click through the second screen, and type the new name in the third. Then look at the new project in the Explorer window to make sure all linked files exist and check the project's Properties for the build configuration and settings (we seem to have two versions of GCC, and the first one, GNU ARM 4.9.3, is old and won't compile the program).

A recap of our progress: We've created a program that runs in the ZWave Application framework. To an empty project we had to add code from the SDK to initialize and boot the devkit board, which needs some small interactions with the framework to get versions. We then added code to start an event handler in the framework, which pulled in many operating system functions as well as ZWave functionality, which we've disabled or removed for the moment until we're done with the hardware. We've added support for buttons and LEDs which brought in another large batch of include files and driver code. Much of the complexity in the build comes from pieces that are scattered throughout the SDK, so we end up with long include paths and many linked source files.

Now we'll access the humidity-temperature (HT) sensor on the devkit board. Communication is done over an I2C bus. According to the sensor's datasheet, its address on the bus is 0x80. There are 16 command bytes to read the sensor, operate the heater, and read product identification bytes. The commands we need are

READ_H_HOLD	0xE5
READ_T_HOLD	0xE3
READ_T_AFTER_H	0xE0
READ_ID2_BYTE1	0xFC
READ_ID2_BYTE2	0xC9

It is slightly more efficient to read both the humidity and temperature together (READ_T_AFTER_H).

We use `I2CSPM_Transfer()` to execute a command. It goes in a loop where we wait while the return value is `i2cTransferInProgress`, or give up after a fixed number of attempts (2). The function takes an `I2C_TransferSeq_TypeDef` data structure, defined in `platform/emlib/inc/em_i2c.h`, which has the destination address, a flag specifying read and/or write operation, and two byte array pointers plus length specifications, one (`buf[0]`) for writing and the other (`buf[1]`) for reading.

To read the sensor in our function [read_HT\(\)](#) the write buffer will contain one byte, the command, and the read buffer must have two. Assuming the read is successful the final return value will be `i2cTransferDone` and we can convert the read buffer into values following the formulas in the datasheet:

$$RH = ((\text{read_byte0} \ll 8) + (\text{read_byte1} \& 0xfc)) - 6000$$

$$T = (((\text{read_byte0} \ll 8) + (\text{read_byte1} \& 0xfc)) * 21965) \gg 13) - 46850$$

The humidity is in milli-percent and must be clipped to the range 0 t/m 100,000. Temperature is in milli-degrees and the formula in the datasheet can overflow 32 bits, so we've scaled constants down by 8.

To read the product information in [verify_sensor\(\)](#) the write buffer will contain two bytes to hold `0xFC` and `0xC9`, and the read buffer must have eight. The first byte returned will be `0x15` for the Si7021 sensor.

In [setup_i2c\(\)](#) we define the pin layout and clocks for the bus by filling in an `I2CSPM_Init_TypeDef` structure and passing it in a call to `I2CSPM_Init()`. None of the `i2cspmconfig.h` files under `${SDK}/hardware/kit/SLW*` has a setup that matches the description in UG381, the devkit spec, so we have to do it ourselves: SCL on PC10 (`I2C0_SCL#14`) which unpacks to Port C, pin 10, location 14; and SDA on PC11 (`I2C0_SDA#16`), or Port C, pin 11, location 16. Setup finishes by calling `verify_sensor()` to check that we can talk to it.

Using the I2C bus means including the header file `i2cspm.h` in our application, so we must

1. Add `${SDK}/hardware/kit/common/drivers` to the C/C++ Include list.
2. Link `${SDK}/platform/emlib/src/em_i2c.c` to hw/.
3. Link `${SDK}/hardware/kit/common/drivers/i2cspm.c` to hw/.
4. Create a file `src/i2cspmconfig.h` which has one define, `I2CSPM_TRANSFER_TIMEOUT 3000000`.

The last file is needed because the devkit board SLWSTK6050A doesn't have an entry under `${SDK}/hardware/kit`.

We enable the sensor with GPIO pin PD15, but this also enables the LCD. That spec requires a periodic pulse on PD13 to avoid damaging the display. We'll use the software timer to meet the datasheet requirements of > 1 Hz, 50% duty cycle pulse, toggling PD13 on and off for 250 ms each. The LCD spec also says to keep `DISP_SCS` on pin PD14 low during `EXTCOMIN`, so we clear the pin. In the application we add two functions, [setup_lcd\(\)](#) which initializes the pins and starts the timer with callback `toggle_extcomin()`.

We can now properly implement sensor reads in our button press [event handler](#). The timer placeholders in `simulate_ht()` get replaced by LED toggles and sensor reads, with the results printed to the UART console. Data logging is done by setting the global number of samples and letting the LCD timer do a sensor read once per

sampling interval for TSAMPLE seconds. The values are cached in global arrays in anticipation of the next step, and are also sent to the UART console.

Serial 0	Serial 1	Admin	Debug
log ID 0	rh 49.427 %	tC 24.954 C	
log ID 1	rh 49.587 %	tC 24.943 C	
log ID 2	rh 49.648 %	tC 24.975 C	
log ID 3	rh 49.580 %	tC 24.954 C	
log ID 4	rh 55.058 %	tC 25.533 C	

Console trace of the start of a logging run.

This version of our application contains these functions:

function	role	description
handle_event	button press	read sensor or start logging
handle_dummy	placeholder	empty ZWave command handler
setup_i2c	sensor read	prepare I2C bus
i2c_strerror	sensor read	string version of I2C status code
verify_sensor	sensor read	confirm HT sensor present
read_HT	sensor read	read humidity and/or temperature
setup_lcd	LCD driver	prepare refresh timer
toggle_extcomin	LCD driver, logger	refresh device, log sensor readings
ApplicationInit	ZAF	pre-task system setup
ApplicationTask	ZAF	post-task setup and event loop
Transport_ApplicationCommandHandlerEx	ZAF	empty ZWave command handler

The project's file list has added two files under hw/ and one under src/.

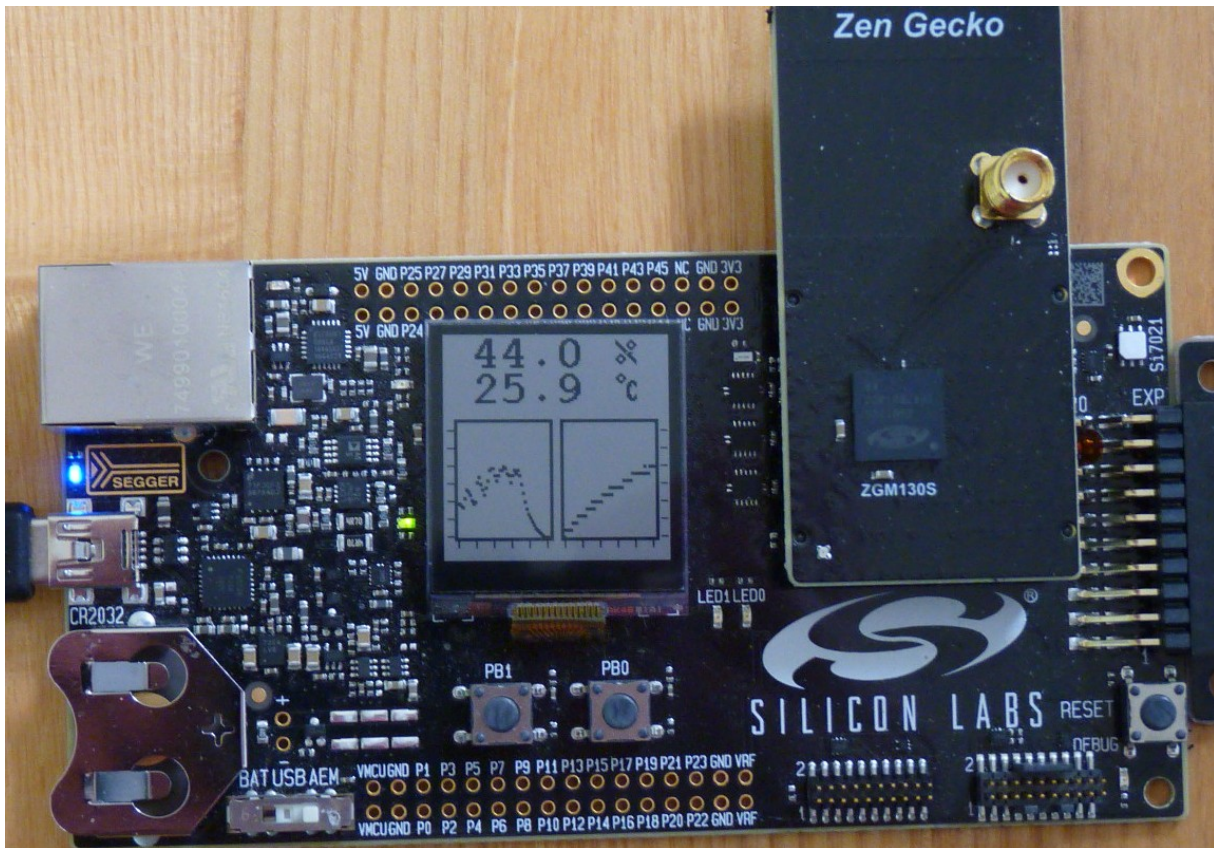
hw/	src/	ZAF_AppUtil/	ZAF_CC/
	config_app.h HTSensor1.c i2cspmconfig.h	ZAF_TSE.c ZW_TransportEndpoint.c	
<i>linked within Simplicity Studio</i>			
em_i2c.c em_ltimer.c gpiointerrupt.c i2cspm.c startup_zgm13.S system_zgm13.c		application_properties.c AppTimer.c board_indicator.c board.c ZAF_command_class_utils.c zaf_event_helper.c ZAF_uart_utils.c ZW_TransportMulticast.c ZW_TransportSecProtocol.c	

Our last hardware change is to [use the LCD](#) to display the humidity and temperature, rather than having to rely on the debug console.

Step 2c: Hardware - LCD

The code for this step is found [here](#).

In addition to eventually sending sensor readings back to the host, we want to use the LCD to show the current values. We'll divide the display into three parts. The top row will show the sensor readings as text. The bottom row will have graph logged values over time. The left graph will contain humidity values, the right temperature.



The DevKit LCD after a log, showing the current readings (top) and graphs of the humidity (below left) and temperature (below right). The sensor is the white square at the top right of the board.

The ZWave SDK provides most, but not all, the files we need to operate the device. There are two headers, one with the display configuration and one with error codes, that are found only in the Gecko SDK. We can use the standard low-level display drivers for the hardware, but will need to modify the text display code to use a font with new characters and will need to write code to display graphs.

To start with the text display,

1. Copy `${SDK}/hardware/kit/drivers/displayfont16x20.h` to `hw/displayfont16x20HT.h` and add/change the four characters we need.
2. Copy `${SDK}/hardware/kit/common/drivers/textdisplay.c` to `hw/`, add references to the new font, and edit `TEXTDISPLAY_New()` and `TextdisplayUpdate()` to allow for fewer text lines to make room for the graph.
3. Copy `${SDK}/hardware/kit/common/drivers/textdisplay.h` to `hw/` and add a member `maxlines` to the `TEXTDISPLAY_Config_t` structure.
4. Copy `gecko_sdk_suite/v2.4/util/silicon_labs/silabs_core/graphics/displayconfigapp.h` to `hw/` and add defines for the new font and to allow VT100 escape codes.
5. Copy `gecko_sdk_suite/v2.4/platform/middleware/glib/em_types.h` into `hw/`.

We need to add four symbols to the font: '-', '.', '%', and degree-C. They need to be in a continuous ASCII series with the other characters, so we put the first three at the start with % replacing the slash (we will have to print a '/' to get the percent, in other words). The degrees sign replaces the semicolon as the next-to-last character, before the space. The characters are bit encoded and were hand-crafted in a spreadsheet and had to be flipped left-right for display. The top of [source file #2](#) defines `FONT_ASCII_START`, `FONT_CHARACTERS`, `FONT_BITS_MASK`, `FONT_BITS_LOG2`, and `fontBits` to describe the new font. The ZWave SDK doesn't contain all the code needed to use the LCD for this devkit, which we find in the Gecko SDK. To [displayconfigapp.h](#) add defines for `INCLUDE_TEXTDISPLAY_SUPPORT`, `TEXTDISPLAY_NUMBER_FONT_16x20HT`, and `INCLUDE_VIDEO_TERMINAL_ESCAPE_SEQUENCE_SUPPORT`, and remove or comment out the real time clock section at the bottom.

To compile we must

1. Link `${SDK}/hardware/kit/common/drivers/display.c` under `hw/`.
2. Link `${SDK}/hardware/kit/common/drivers/displayls013b7dh03.c` under `hw/`.
3. Link `${SDK}/hardware/kit/common/drivers/displaypalemilib.c` under `hw/`.
4. Add `${SDK}/hardware/kit/common/bsp` to the C/C++ Include list.
5. Link `${SDK}/platform/emlib/src/em_prs.c` under `hw/`.

We are now ready to display the sensor values in [display_humidity\(\)](#) or `display_temperature()`. We use VT100 escape codes to get to the right line and print values digit by digit, dropping leading zeroes, adding a decimal point when appropriate, and finishing with a space and the units. There are separate routines for humidity and temperature since the initial positions are different and we must handle negative temperatures. Escape codes are written to screen with `TEXTDISPLAY_WriteString()`, characters with `TEXTDISPLAY_WriteChar()`. You'll find the escape codes defined in `textdisplay.h`. Note that we send '/' or ';' for the units, per our font substitution.

The driver function `DISPLAY_Init()` replaces the GPIO pin setup we had to do in Step 2b, and we no longer need to use the software timer to drive the EXTCOMIN pin. We create the [appconfig](#) structure to store the display configuration in a global variable which is passed to each `TEXTDISPLAY_*` function, and replace the `setup_lcd()` function in Step 2b with [setup_display\(\)](#). This must be called before `setup_i2c()` because it enables GPIO pin 15, the display and the sensor.

In our event handler we display the sensor reading after printing it to the UART console.

Our graphics driver will be modeled on `textdisplay.c`. The public interface will include `GRAPHDISPLAY_New()` to set up the geometry of the area and draw boxes around the graphs, `GRAPHDISPLAY_Delete()` to release any resources we've reserved, `GRAPHDISPLAY_Clear()` to turn off all the pixels inside the frame (remembering that the display is inverted, or black-on-white, so 'off' really means 'white' and is a set bit), `GRAPHDISPLAY_GetWidth()` to get the size of the drawing area, and `GRAPHDISPLAY_SetPoint()` to turn a pixel on or off. For batch changes, the drawing functions have a `redraw` argument that if false sets pixels in the backing store without sending the whole store to the display. Only for the last point should `redraw` be set true. Pixels are bit-packed into the store. The workhorse function underneath this code is `pPixelMatrixDraw()`, part of the `displayls013b7dh03.c` code. The driver is in `graphdisplay.c`, the header is `graphdisplay.h`, and both are in the `hw/` directory.

Clearing the text, which sets the update mode to `TEXTDISPLAY_UPDATE_MODE_FULL`, erases the entire display, so we need to modify the text display so it thinks there are fewer lines on the LCD than there really are. We add a member `maxlines` to the display configuration structure to provide the limit (0 will still use the entire screen), setting it before calling `TEXTDISPLAY_New()`. The driver will work without changes with the overridden line count, except for one edit to `TextdisplayUpdate` to keep the default behavior of clearing the bottom of the screen automatically when we haven't set `maxlines`.

We need to re-organize the sensor log before we can graph the data. We gather the history of the measurements in the [htsamples](#) structure, also using it to track the range of values, position in the graph, and the sampling time and count. We don't want to use a fixed range for graphing so we can adapt to the readings, be they fairly stable over a short run or with large swings over a longer. We add two functions: `clear_log()` to initialize everything, and `start_sampling()` to prepare the log and graphs and start the timer. We change the timer callback to store the readings in the structure, updating the range in the process, and displaying/graphing the point. Remember this callback no longer contains the LCD `EXTCOMIN` toggle. We use separate functions to graph each value (with some work we could abstract the log so we would need only one function, because the two are mostly the same). If the vertical scale changes we must clear the graph and redraw all points, otherwise we can just plot the new point. The logic is straightforward within each function.

Our application now contains these new or modified functions. The I2C and ZAF functions `setup_i2c()`, `i2c_strerror()`, `verify_sensor()`, `read_HT()`, `handle_dummy()`, `ApplicationInit()`, `ApplicationTask()`, and `Transport_ApplicationCommandHandlerEx()` are unchanged.

function	role	description
handle_event	button press	read sensor or start logging
start_sampling	logger	clear log/graph, start timer
sample_ht	logger	store and display/graph sensor values
clear_log	logger	prepare data store for logging
setup_display	LCD driver	initialize text, graph displays
display_humidity	LCD driver	show humidity in text display
display_temperature	LCD driver	show temperature in text display
clear_graphs	LCD driver	erase data from graph display
graph_humidity	LCD driver	draw humidity log in graph
graph_temperature	LCD driver	draw temperature log in graph
draw_2x2point	LCD driver	draw 2x2 point in a graph

We've had to add many files to the hardware directory.

hw/	src/	ZAF_AppUtil/	ZAF_CC/
displayconfigapp.h displayfont16x20HT.h em_types.h graphdisplay.c graphdisplay.h textdisplay.c testdisplayconfig.h textdisplay.h	config_app.h HTSensor1.c i2cspmconfig.h	ZAF_TSE.c ZW_TransportEndpoint.c	
<i>linked within Simplicity Studio</i>			
display.c displayls013b7dh03.c displaypalemlib.c em_i2c.c em_letimer.c em_prs.c gpinterrupts.c i2cspm.c startup_zgm13.S system_zgm13.c		application_properties.c AppTimer.c board_indicator.c board.c ZAF_command_class_utils.c zaf_event_helper.c ZAF_uart_utils.c ZW_TransportMulticast.c ZW_TransportSecProtocol.c	

The picture at the start of this section shows the display after a logging session. We've implemented all the hardware functions we wanted, so it's time to turn to [ZWave](#).

Step 3a: ZWave - Inclusion and Exclusion

The code for this step is found [here](#).

The hardware is working, it's time to add ZWave support. Our primary goal is to implement CC_SENSOR_MULTILEVEL for both the humidity and temperature channels. We'll implement Version 11 of the command class, with GET/REPORT and SUPPORTED_GET/REPORT for sensors and scales. We'll use CC_CONFIGURATION to control the logger, supporting three parameters: #1, a 4 byte integer with the time between samples in seconds (the size chosen to allow 86400, or once per day); #2, a 1 byte integer with the number of samples (0 to run without stopping); and #3, a 1 byte start/stop (non-zero/zero) flag. We'll store at most 64 samples, based on the width of the LCD graphs, so we can statically allocate arrays and the back stores for the graphs. Logging will not persist over a power loss, so we don't need to worry about storing state in NVM, and we'll ignore configuration changes while the logger is running.

We are not trying to build a device that will pass certification. We will be ignoring many required command classes, including ZWave+, Indicator, Association, Basic, Device Reset, Firmware Update, and Supervision. We will not implement Smart Start, nor S2 Security. For reference, ZWave+ requires us to use GENERIC_TYPE_SENSOR_MULTILEVEL and SPECIFIC_TYPE_ROUTING_MULTILEVEL_SENSOR, and Sensor Multilevel instead of Basic. It also requires Association, Association Group Info, Device Reset, Firmware Update Meta-Data, Indicator, Manufacturer Specific, Multi-Channel Association, Power Level, S2, Supervision, Transport Service, Version, and Zwave Plus Info; Multi-Command would be recommended. Many of these are easily supported using the full framework, which we'll discuss in Step 4.

As a first step, implementing inclusion and exclusion is enough work, as it will force us to deal with the application framework parts we simplified back in [Step 2a](#). You'll remember we removed association groups from the ZAF_TSE.c source file and multichannels from ZW_TransportEndpoint.c. We linked in many files to use them unchanged, including application_properties.c, AppTimer.c, board_indicator.c, board.c, ZAF_command_class_utils.c, zaf_event_helper.c, ZAF_uart_utils.c, ZW_TransportMulticast.c, and ZW_TransportSecProtocol.c. All these should be under the ZAF_AppUtil/ directory.

We start by cleaning up the configuration files for the framework. Include ZW_product_id_enum.h in src/config_app.h and remove the _PRODUCT_TYPE_ID_ENUM_ type. Move MAX_TXPWR and MEASURED_0DBM_TXPWR into src/config_rf.h, include ZW_radio_api.h, and define RADIO_REGION as REGION_US, also replacing the use of REGION_US in the application code. In ApplicationInit() we need to start the radio by calling BRD420xBoardInit(). Call this, and setup_display(), after starting the UART, as it includes debug prints that we don't want to lose. (The initialization function from [Step 3b](#) shows these changes, as well as edits for the NIF.)

When we created our event handlers in Step 2a, we left two placeholders that we now have to implement. The ZwCommandStatusQueue will handle events from the framework to the application, and ZWRxQueue will handle incoming frames. There is also a pZwTxQueue for outbound frames and pZWCommandQueue for commands to the framework, neither of which we'll use. Events in the command status queue have payloads of type SZwaveCommandStatusPackage; they are processed in [handle_cmdstatus\(\)](#). Its first byte/member, called eStatusType, is an enumeration EZwaveCommandStatusType. The second member, called Content, is a type-specific structure of varying length. For example and assuming the payload variable is called 'status', during inclusion we'll get a EZWAVECOMMANDSTATUS_LEARN_MODE_STATUS packet for status.eStatusType with the detailed state in status.Content.LearnModeStatus.Status. Frame events, processed in [handle_frame\(\)](#), also have two members in their structures, the first called eReceiveType and the second with type-specific details

called `uReceiveParams`. All of these are defined in `${SDK}/ZWave/API/ZW_application_transport_interface.h`. The event handlers will switch on the first member, either `eStatusType` or `eReceiveType`, and then process the details in the second. For this step, we do not need to do anything with frame events, and only must consider `EZWAVECOMMANDSTATUS_LEARN_MODE_STATUS` and `EZWAVECOMMANDSTATUS_NETWORK_LEARN_MODE_START`.

The demo programs define five states for the application. `STARTUP` means they are setting up non-volatile memory. During `INIT` they read what's stored in NVM, set up association groups, start the event scheduler, and begin Smart Start scanning. While `IDLE` they handle button presses, either entering learn mode or sending the NIF, or sending a message to the lifeline. They mark `LEARN_MODE` with LEDs and allow canceling the operation. `RESET` does just that, for the board and NVM.

We will simplify this. Our application state will contain a flag indicating if we're in learn mode or not. Progress will translate into an enumeration `LEARN_[EXCLUDED|INCLUDED|UNDERWAY|FAILED]` summary which we will pass to a `set_indicator()` function to control LED1, which we set aside back in Step 2a for this purpose. The LED will turn on if included, blink slowly while underway, and blink quickly for a while after a failure. Both the flag and LED are set in `handle_cmdstatus()`.

While learning our button press event handler will ignore `Button1` events. Otherwise it will call `ZAF_setNetworkLearnMode()` depending on the current inclusion state (available as `app->pNetworkInfo->eInclusionState` where `app` is cached and retrieved from `ZAF_getAppHandle()`). See [handle_event\(\)](#).

These modifications require surprisingly few changes to files linked to the project; just getting a project to compile within the framework, without using it, has pulled in most of the dependencies. There are no changes to the project's (build) properties.

1. Link `${SDK}/ZAF/ApplicationUtilities/board_BRD420x.c` under `ZAF_AppUtil/`.
2. Link `${SDK}/ZAF/ApplicationUtilities/ZAF_network_learn.c` under `ZAF_AppUtil/`.

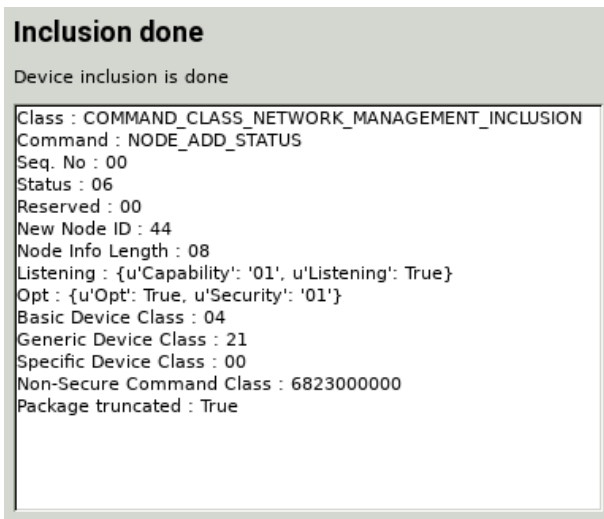
Only a handful of functions change for this functionality. Any not listed here are the same as in Step 2c.

function	status	role	description
<code>handle_event</code>	edit	button press	implement button 1
<code>handle_dummy</code>	remove		
<code>set_indicator</code>	new	LED1	show learn status on LED
<code>ApplicationInit</code>	edit	ZAF	use <code>config_rf.h</code> , init radio
<code>ApplicationTask</code>	edit	ZAF	extra handlers
<code>handle_frame</code>	new	ZAF	frame event handler
<code>handle_cmdstatus</code>	new	ZAF	command status event handler

The directory listing has just three new entries, the RF configuration file and the two ZAF source links.

hw/	src/	ZAF_AppUtil/	ZAF_CC/
displayconfigapp.h displayfont16x20HT.h em_types.h graphdisplay.c graphdisplay.h textdisplay.c testdisplayconfig.h textdisplay.h	config_app.h config_rf.h HTSensor1.c i2cspmconfig.h	ZAF_TSE.c ZW_TransportEndpoint.c	
<i>linked within Simplicity Studio</i>			
display.c displayls013b7dh03.c displaypalemlib.c em_i2c.c em_letimer.c em_prs.c gpoinerrupts.c i2cspm.c startup_zgm13.S system_zgm13.c		application_properties.c AppTimer.c board_BRD420x.c board_indicator.c board.c ZAF_command_class_utils.c zaf_event_helper.c ZAF_network_learn.c ZAF_uart_utils.c ZW_TransportMulticast.c ZW_TransportSecProtocol.c	

With this we have the basic functionality we wanted. If you press Button1 without having a controller waiting for the node, you will eventually see a failure indicator. If there is a controller waiting, you'll see a success indicator and console message. The controller will see the some correct values in the NIF, including the right device classes, and some incorrect, like the command class list. We'll fix that [next](#). Sending commands from the controller to the devkit will do nothing, but you will see console messages for the framework events it generates.



zipgateway information after inclusion. Command classes 0x68 (ZIP Naming) and 0x23 (ZIP) are added by the gateway. Device classes are correct.

27 Jul 20	08:56:14.025	SOF	RX RES	GetNodeProtoInfo				06 01 09 01 41 d3 9c 01 04 21 00 dd
27 Jul 20	08:56:14.025	SOF	TX REQ	ReqNodeInfo				06 01 05 00 60 44 01 df
27 Jul 20	08:56:14.032	SOF	RX RES	ReqNodeInfo				06 01 04 01 60 01 9b
27 Jul 20	08:56:14.032	ACK	TX					06
27 Jul 20	08:56:14.050	SOF	RX REQ	AppUpdate				01 09 00 49 84 44 03 04 21 00 59
27 Jul 20	08:56:14.050	ACK	TX					06
27 Jul 20	08:56:17.477	SOF	TX REQ	AddNode				01 05 00 4a 05 00 b5
27 Jul 20	08:56:17.480	ACK	RX					06


```

[27 Jul 20 08:56:14.050] < 01 09 00 49 84 44 03 04 21 00 59
0x01  SOF
9      = size
0x00  REQ
      SerialCommand :
0x49  AppUpdate
      AppState :
0x84  HaveNodeInfo
68    = node
3      = count
0x04  BasicType - RoutingSlave
0x2100 Generic/SpecificType - MultiSensor/NotUsed
0x59  = checksum

```

```

[27 Jul 20 08:56:14.025] < 06 01 09 01 41 d3 9c 01 04 21 00 dd
0x06  ACK
0x01  SOF
9      = size
0x01  RES
      SerialCommand :
0x41  GetNodeProtoInfo
0x80  ListenSupport
1      = routing
2      = maxspeed
3      = protocolver
0x80  OptFnSupport
1      = can_beam
1      = routing_slave
1      = specific_device
0      = controller
0      = security
0x01  = reserved
0x04  BasicType - RoutingSlave
0x2100 Generic/SpecificType - MultiSensor/NotUsed
0xdd  = checksum

```

zpiffer trace of inclusion. AppUpdate in left (blue) window includes no command classes. Other NIF components in right (red) window.

Step 3b: ZWave - NIF and CC_VERSION

The code for this step is found [here](#).

Our application in Step 3a sends a NIF with a bogus command class list. We want to fix that, and, since we haven't created any commands, will add one, CC_VERSION. This means we need to finish setting up the framework, figuring out how frames are routed through the event handlers to interact with the command class code. We then build the correct command class list and provide it to ZAF.

Incoming frames arrive in two places. The framework leaves a stub that the transport layer calls, `Transport_ApplicationCommandHandlerEx()`, for command classes that it has already unpacked. The frame handler stores the frame as the payload to a `SZwaveReceivePackage` structure, but does not unpack it. There are separate ways of processing the data.

An unpacked command is put in a `ZW_APPLICATION_TX_BUFFER` structure, whose first two members are the command class and command bytes. The third member is a structure particular to the command, and the type of this member is a union of all possibilities. `${SDK}/ZWave/API/ZW_classcmd.h` defines these. `ZW_VERSION_REPORT_2BYTE_V2_FRAME`, for example, has eleven more bytes with the library type, protocol version, firmware version, hardware version, and up to two additional firmware targets.

`Transport_ApplicationCommandHandlerEx()` is passed this structure, along with the receive options flags and frame size. It switches on the command class byte and calls the command class directly. The ZAF command classes call this a handler.

The incoming frame handler pulls the frame off the ZwRxQueue, of type SZwaveReceivePackage, defined in `{SDK}/ZWave/API/ZW_application_transport_interface.h`. The data is identified with an EZwaveReceiveType enumeration, with values for SINGLE, MULTI, NODE_UPDATE, and SECURITY_EVENT. The second member, uReceiveParams, is a union of structures, one per type. For SINGLE the structure includes the payload, its length, and the receive options flags; MULTI also adds a node mask. NODE_UPDATE is for controllers, and SECURITY_EVENT for a specific S2 need. The payload has not been unpacked, so ZAF includes another notification system called the CommandPublisher to do that. The publisher knows about command classes by some trickery during the build: the command class source file uses the REGISTER_CC() macro to create a global variable that's added to an array in a named section of memory (called the HANDLER_SECTION) during compilation. The publisher can loop through the array to find the command class and call the corresponding receive handler after unpacking the frame; it can also pick a handler from the command byte.

The application's [handle_frame\(\)](#) pulls a frame off the queue and switches on the received type, where we need only support SINGLE. It hands off the payload to the CommandPublisher which does the unpacking and calls the appropriate function in the appropriate command class. The application source code contains a comment showing how you can start to pick apart the payload if you wish.

ZAF needs to know which command classes the application actually supports, which may differ from those available through the publisher. It expects up to three byte arrays with the CC lists, one for unsecured commands, a second for those that can be accessed unsecured even if the node is included securely, and a third for secured. These arrays and their length go in a [global variable](#) of type app_node_information_t, along with the device options mask we've put in src/config_app.h (set to APPLICATION_NODEINFO_LISTENING) and the node's generic and specific type, also defined in src/config_app.h. For this step we're just going to support Version unsecured, and will pass empty pointers for the other two options; we'll see how to include secure commands in Step 3e. This NIF is passed to Transport_OnApplicationInitSW() at the start of the application task.

Confusingly, the framework actually contains two versions of the NIF, in two different structures. In addition to app_node_information_t, there is also a SAppNodeInfo_t, which is stored in a second global variable appnif2 and passed to ZW_ApplicationRegisterTask() in the [initialization function](#). The two structures contain the same information, just with different names and in a different order (arrays and lengths are swapped). We add a function copy_nif() to transform between the two.

To put all this simply, adding a command class requires:

- implementing the command, creating a handler for the incoming packet
- registering the handler with the command publisher
- calling the publisher in the frame event handler
- providing an application command handler for the transport layer that, based on the command class, will call the command class handler directly
- adding the command class byte to the unsecured/available unsecured/secured list in the NIF
- copying the NIF to the second format and storing that with the ZWave configuration, provided when starting the application's FreeRTOS task
- registering the NIF with the transport layer

OK, maybe that's not so simple. It's certainly repetitive. If you compare the changes to the application source code between Step 3a and Step 3b, you'll see this is all that's happened (plus one addition for Version, described below). It requires several changes to the project's Properties and files:

1. Removing `${SDK}/ZAF/CommandClasses/Common` from the C/C++ Include list.
2. Linking on the file system `${SDK}/ZAF/CommandClasses/Common/CC_Common.h` under `ZAF_CC/`.
3. Linking on the file system `${SDK}/ZAF/CommandClasses/Version/CC_Version.h` under `ZAF_CC/`.
4. Linking on the file system `${SDK}/ZAF/CommandClasses/Supervision/CC_Supervision.h` under `ZAF_CC/`.
5. Linking on the file system `${SDK}/ZAF/CommandClasses/MultiChan/multichannel.h` under `ZAF_CC/`.
6. Linking `${SDK}/ZAF/ApplicationUtilities/ZAF_tx_mutex.c` under `ZAF_AppUtil/`.
7. Linking `${SDK}/ZAF/CommandClasses/Supervision/CC_Supervision.c` under `ZAF_CC/`.
8. Removing our edited `ZAF_AppUtil/ZW_TransportEndpoint.c`.
9. Linking `${SDK}/ZAF/ApplicationUtilities/ZW_TransportEndpoint.c` under `ZAF_AppUtil/`.
10. Adding `#define NUMBER_OF_ENDPOINTS 0` to `src/config_app.h`.

It is important in steps #2-5 that the links be created on disk and not within Simplicity Studio, otherwise the header files will not physically exist and the build will fail. This is not true for source files as in #6 and #7, which correctly translate into SDK references when the makefile is built. This project layout is a preference for us, to keep things in a concise file hierarchy and to shorten the include path and compiler command line. There are two alternatives Silicon Labs prefers, based on the demo programs. You could instead add each command class directory to the C/C++ Include list, or you could use project modules within Simplicity Studio. On the project Properties, the build section contains an entry called Project Modules, and here you can check which commands you want to use. Simplicity Studio will then make a local copy on disk from the SDK, one directory per command class.

Bringing up the Version command requires a few more changes. The command itself is built into the framework and we don't need to start from scratch. That implementation leaves three functions as stubs, which we must supply in our [ZAF/CC_Version.c](#): `CC_Version_getNumberOfFirmwareTargets()`, `CC_Version_GetFirmwareVersion_handler()`, and `CC_Version_GetHardwareVersion_handler()`. We fill in dummy values for each of these. The source code requires the corresponding header which was linked in #3 above. Finally, during the application's initialization we need to call `CC_Version_SetApplicationVersionInfo()` with the values we've defined in `src/config_app.h`.

At this time we've undone the simplifications back in Step 2a to get the framework set up, except for the edits to `ZAF_TSE.c` to avoid dealing with association groups. The remaining source code edits deal with setting up the NIF and command routing and implementing the missing pieces to the `CC_Version` functionality that's built in to the framework.

object	status	role	description
appnif1 (global var)	new	transport	NIF for transport layer
ApplicationInit	edit	ZAF	set NIFs, version information
ApplicationTask	edit	ZAF	register transport layer NIF
Transport_AppCmdHandlerEx	new	transport	transport layer command router
handle_frame	edit	ZAF	pass frame to command publisher
copy_nif	new	ZAF	copy transport NIF to protocol NIF
ZAF_CC/Version.c	new	command	add missing functions

In the file listing we've made ZW_TransportEndpoint.c a Simplicity Studio link and populated the command class directory.

hw/	src/	ZAF_AppUtil/	ZAF_CC/
displayconfigapp.h displayfont16x20HT.h em_types.h graphdisplay.c graphdisplay.h textdisplay.c testdisplayconfig.h textdisplay.h	config_app.h config_rf.h HTSensor1.c i2cspmconfig.h	ZAF_TSE.c	CC_Version.c
<i>linked within Simplicity Studio</i>			
display.c displayls013b7dh03.c displaypalemlib.c em_i2c.c em_letimer.c em_prs.c gpiointerrupts.c i2cspm.c startup_zgm13.S system_zgm13.c		application_properties.c AppTimer.c board_BRD420x.c board_indicator.c board.c ZAF_command_class_utils.c zaf_event_helper.c ZAF_network_learn.c ZAF_uart_utils.c ZW_TransportEndpoint.c ZW_TransportMulticast.c ZW_TransportSecProtocol.c	CC_Supervision.c multichannel.c
<i>linked on file system</i>			
			CC_Common.h CC_Supervision.h CC_Version.h multichannel.h

Sending VERSION_GET will return the right information. The command list has 0x86 (Version) and we see the correction application and library versions. Now we can implement the [sensor class](#).

Inclusion done

Device inclusion is done

```
Class : COMMAND_CLASS_NETWORK_MANAGEMENT_INCLUSION
Command : NODE_ADD_STATUS
Seq. No : 00
Status : 06
Reserved : 00
New Node ID : 45
Node Info Length : 09
Listening : {u'Capability': '01', u'Listening': True}
Opt : {u'Opt': True, u'Security': '01'}
Basic Device Class : 04
Generic Device Class : 21
Specific Device Class : 01
Non-Secure Command Class : 866823000000
Package truncated : True
```

zipgateway information after inclusion. Command classes now include 0x86 (Version).

27 Jul 20	08:58:50.410	SOF	RX RES	GetNodeProtoInfo					06 01 09 01 41 d3 9c 01 04 21 01 dc
27 Jul 20	08:58:50.410	SOF	TX REQ	ReqNodeInfo					06 01 05 00 60 45 01 de
27 Jul 20	08:58:50.417	SOF	RX RES	ReqNodeInfo					06 01 04 01 60 01 9b
27 Jul 20	08:58:50.417	ACK	TX						06
27 Jul 20	08:58:50.437	SOF	RX REQ	AppUpdate					01 0a 00 49 84 45 04 04 21 01 86 db
27 Jul 20	08:58:50.437	SOF	TX REQ	SendDataBridge	255	69	CC_Version	GetICC	06 01 0f 00 a9 ff 45 03 86 13 86 25 00 00 00 01 d7
27 Jul 20	08:58:50.447	SOF	RX RES	SendDataBridge					06 01 04 01 a9 01 52
27 Jul 20	08:58:50.447	ACK	TX						06
27 Jul 20	08:58:50.461	SOF	RX REQ err	SendDataBridge					01 18 00 a9 01 00 00 01 00 7e 71 71 71 00 00 03 00 00 00 05 01 00 00 37
27 Jul 20	08:58:50.461	ACK	TX						06

[27 Jul 20 08:58:50.437] < 01 0a 00 49 84 45 04 04 21 01 86 db	[27 Jul 20 08:58:50.410] < 06 01 09 01 41 d3 9c 01 04 21 01 dc
0x01 SOF	0x06 ACK
10 = size	0x01 SOF
0x00 REQ	9 = size
SerialCommand :	0x01 RES
0x49 AppUpdate	SerialCommand :
AppState :	0x41 GetNodeProtoInfo
0x84 HaveNodeInfo	0x80 ListenSupport
69 = node	1 = routing
4 = count	2 = maxspeed
0x04 BasicType - RoutingSlave	3 = protocolver
0x2101 Generic/SpecificType - MultiSensor/Routing	0x80 OptFnSupport
0x86 CommandClass - CC_Version	1 = can_beam
0xdb = checksum	1 = routing_slave
	1 = specific_device
	0 = controller
	0 = security
	0x01 = reserved
	0x04 BasicType - RoutingSlave
	0x2101 Generic/SpecificType - MultiSensor/Routing
	0xdc = checksum

zipiffer trace of inclusion. AppUpdate in left (blue) window shows CC_Version.

Date	Time	Frame	Flags	Serial Command	source	destination	Command Class	Command
27 Jul 20	08:58:50.437	SOF	RX REQ	AppUpdate				
27 Jul 20	08:58:50.437	SOF	TX REQ	SendDataBridge	255	69	CC_Version	GetCC
27 Jul 20	08:58:50.447	SOF	RX RES	SendDataBridge				
27 Jul 20	08:58:50.447	ACK	TX					
27 Jul 20	08:58:50.461	SOF	RX REQ err	SendDataBridge				
27 Jul 20	08:58:50.461	ACK	TX					
27 Jul 20	08:58:50.467	SOF	RX REQ	AppCmdBridge	69	1	CC_Version	CCReport
27 Jul 20	08:58:50.467	SOF	TX REQ	SendDataBridge	255	69	CC_Version	GetCapability
27 Jul 20	08:58:50.476	SOF	RX RES	SendDataBridge				
27 Jul 20	08:58:50.476	ACK	TX					
27 Jul 20	08:58:50.489	SOF	RX REQ err	SendDataBridge				
27 Jul 20	08:58:50.489	ACK	TX					
27 Jul 20	08:58:50.494	SOF	RX REQ	AppCmdBridge	69	1	CC_Version	CapabilityReport
27 Jul 20	08:58:50.494	SOF	TX REQ	SendDataBridge	255	69	CC_Version	GetZW
27 Jul 20	08:58:50.504	SOF	RX RES	SendDataBridge				
27 Jul 20	08:58:50.504	ACK	TX					
27 Jul 20	08:58:50.518	SOF	RX REQ err	SendDataBridge				
27 Jul 20	08:58:50.518	ACK	TX					
27 Jul 20	08:58:50.528	SOF	RX REQ	AppCmdBridge	69	1	CC_Version	ZWReport
27 Jul 20	08:58:50.528	ACK	TX					


```

[27 Jul 20 08:58:50.494] < 01 0c 00 a8 00 01 45 03 86 16 07 00 7e f5
0x01 SOF
12 = size
0x00 REQ
SerialCommand :
0xa8 AppCmdBridge
0x00 RxStatus - Single
1 = destnode
69 = srcnode
3 = size
CommandClass :
0x86 CC_Version
0x16 CapabilityReport
1 = have_zwsw_ver
1 = have_cc_ver
1 = have_ver_ver
0 = size
0x7e SignalStrength - Saturated
0xf5 = checksum

```

```

CommandClass :
0x86 CC_Version
0x18 ZWReport
7 = sdk_maj
12 = sdk_min
1 = sdk_patch
3 = zaf_maj
2 = zaf_min
0 = zaf_patch
31 = zaf_build
0 = host_maj
0 = host_min
0 = host_patch
0 = host_build
7 = zw_maj
12 = zw_min
1 = zw_patch
31 = zw_build
3 = app_maj
2 = app_min
0 = app_patch
0 = app_build
0 = size
0x7e SignalStrength - Saturated

```

zpiffer shows further probing of CC_Version during inclusion, including command class capabilities (left/blue) and ZWave and app versions (right/red). The SDK and ZW fields match the library version, the app fields match config_app.h.

Step 3c: ZWave - CC_SENSOR_MULTILEVEL

The code for this step is found [here](#).

Adding ZWave access to the sensor will be pretty simple. We need to put the CC byte in the NIF, route the packet in the transport layer's application command handler, and implement the command class. Complicating things will be the need to transmit a reply, and we'll try to make our implementation general, using stubs that the application must provide for specific information like supported sensor types and scales. A diff of the application source code to Step 3b will also show many code clean-ups that we won't discuss.

SDS13812 defines sensor types. Air temperature sensors have value (ID) 0x01, with the supported bit mask being bit 0 of byte 1. We will provide two scales, Fahrenheit (ID 0x01) and Celsius (0x00). Humidity sensors

have value 0x05, with the supported mask being bit 5 of byte 1. There are two possible scales, relative (ID 0x00) and absolute (ID 0x01), and we'll only support the first. Both sensor readings are in milli-percent/degrees, so the values reported will use 4 bytes with a precision of 3. This means that the combined mask for `SENSOR_MULTILEVEL_SUPPORTED_SENSOR_REPORT` is 0x11. `SENSOR_MULTILEVEL_SUPPORTED_SCALE_REPORT` will return 0x6C for temp (F) and 0x64 for humidity and temp (C).

Structures holding the contents of the frames we'll use are defined in `{SDK}/ZWave/API/ZW_classcmd.h`. We will use the latest version of the command class, V11. The frames are `ZW_SENSOR_MULTILEVEL_GET_V11_FRAME`, `ZW_SENSOR_MULTILEVEL_REPORT_[1234]BYTE_V11_FRAME`, `ZW_SENSOR_MULTILEVEL_SUPPORTED_GET_SENSOR_V11_FRAME`, `ZW_SENSOR_MULTILEVEL_SUPPORTED_SENSOR_REPORT_[1234]BYTE_V11_FRAME`, `ZW_SENSOR_MULTILEVEL_SUPPORTED_GET_SCALE_V11_FRAME`, and `ZW_SENSOR_MULTILEVEL_SUPPORTED_SCALE_REPORT_V11_FRAME`. The reports have four different versions depending on the number of mask bytes that must be sent; we choose the appropriate version for the value.

Outgoing messages use three structures, `TRANSMIT_OPTIONS_TYPE_SINGLE_EX` defined in `{SDK}/ZAF/ApplicationUtilities/ZW_TransportEndpoint.h`, `ZAF_TRANSPORT_TX_BUFFER` defined in `{SDK}/ZAF/ApplicationUtilities/ZAF_tx_mutex.h`, and `ZAF_APPLICATION_TX_BUFFER` defined in `ZW_classcmd.h`. The first has the source and destination nodes, security flags, and transmission flags. The preferred way to fill it in is to use the `RxToTxOptions()` function to convert a corresponding structure on the incoming GET. If we send a reading in response to a button press or during logging, then for this demo we force the destination to be node 1, the controller (in a full product you would send it to the lifeline). The second structure contains multichannel endpoint information, a `SUPERVISION_GET` frame, and the actual transmission stored in the third structure. It is a union of structures of all possible frame contents, like those in the previous paragraph. We fill in the correct frame with the sensor value or supported bit mask and zero the supervision structure. Call `Transport_SendResponseEP()` with the options and transport buffer to send the report.

The transport layer has a second queue for outgoing frames, accessed with `Transport_SendRequestEP()`. Since requests must be used for unsolicited messages, we need to use it for values sent from button presses. However, it seems to have no difference in the packet sent (you'll notice the frame type is REQ in the zpiiffer traces) and in the transport layer code there are only a couple of differences: requests override the source node for multichannel frames, and the callbacks after transmission have different signatures, a request getting a transmission result and a response the transmit status.

To implement the command class, create `ZAF_CC/CC_MultiSensor.[ch]`. We'll have two functions for our ZWave functionality, [handleCCMultilevelSensor\(\)](#) for incoming frames and [sendSensorValue\(\)](#) for outgoing reports. All other transmissions are REPORTs in response to GETs and are performed in the handler. The source code contains these two functions, plus an internal helper with the common send code for both report paths. It also invokes the `REGISTER_CC` macro to add the handler to the command publisher. The handler is called in the application by `Transport_ApplicationCommandHandlerEx()` (the command publisher is already doing the routing in the frame handler), and values are sent in response to button press events.

The command class will also prototype five functions that must be implemented in the application to supply

device-specific information. This architecture is similar to that used by the binary switch and user code commands already included in ZAF. [appSensorValidType\(\)](#) validates the sensor type passed in a GET. [appSensorMask\(\)](#) generates the bit mask for the supported sensor types, and [appSensorScaleMask\(\)](#) the scale specification for a sensor type. [appSensorLevel\(\)](#) returns the precision/scale/size byte for a sensor reading. [appSensorValue\(\)](#) returns the humidity or temperature after reading it from the sensor. It does the Fahrenheit conversion locally.

The differences in the application code to Step 3b, ignoring edits for code cleanup, are

object	status	role	description
appnif1 (global var)	edit	transport	add command class
sample_ht	edit	logger	send reading over ZWave
send_temperature humidity	new	ZWave	send reading over ZWave
Transport_AppCmdHandlerEx	edit	transport	call command class handler
handle_cmdstatus	edit	ZAF	callback after successful Tx
am_included	new	ZWave	test if node included in a network
appSensorValidType	new	CC support	validate requested sensor
appSensorMask	new	CC support	get mask for supported sensors
appSensorScaleMask	new	CC support	get mask for supported scales
appSensorValue	new	CC support	read sensor
appSensorLevel	new	CC support	get size of sensor value

An edit to [handle_cmdstatus\(\)](#) is needed to avoid filling up a limited queue of callbacks after frames are transmitted. Comments in ZW_TransportEndpoint.c indicate the queue is supposed to be a ring, but it doesn't look like it cycles correctly and the head eventually collides with the tail. Executing the callback on a EZWAVECOMMANDSTATUS_TX event solves the problem.

The project files now include the source for the new command class.

hw/	src/	ZAF_AppUtil/	ZAF_CC/
displayconfigapp.h displayfont16x20HT.h em_types.h graphdisplay.c graphdisplay.h textdisplay.c testdisplayconfig.h textdisplay.h	config_app.h config_rf.h HTSensor1.c i2cspmconfig.h	ZAF_TSE.c	CC_MultiSensor.c CC_MultiSensor.h CC_Version.c
<i>linked within Simplicity Studio</i>			
display.c displayls013b7dh03.c displaypalemlib.c em_i2c.c em_letimer.c em_prs.c gpinterrupts.c i2cspm.c startup_zgm13.S system_zgm13.c		application_properties.c AppTimer.c board_BRD420x.c board_indicator.c board.c ZAF_command_class_utils.c zaf_event_helper.c ZAF_network_learn.c ZAF_uart_utils.c ZW_TransportEndpoint.c ZW_TransportMulticast.c ZW_TransportSecProtocol.c	CC_Supervision.c multichannel.c
<i>linked on file system</i>			
			CC_Common.h CC_Supervision.h CC_Version.h multichannel.h

Testing, you'll find the application now responds to all CC_SENSOR_MULTILEVEL commands. Buttons 3 and 4 will send single sensor readings, and button 2 will start logging with periodic updates sent to the controller. The setup of the log is still hard-coded, and we need to change that next with the [Configuration class](#).

<pre> [27 Jul 20 09:44:16.316] > 01 10 00 a9 03 46 04 31 04 05 00 25 00 00 00 00 08 1a 0x01 SOF 16 = size 0x00 REQ SerialCommand : 0xa9 SendDataBridge 3 = srcnode 70 = destnode 4 = size CommandClass : 0x31 CC_MultiSensor 0x04 Get 0x05 Sensor - Humidity 0 = scale TxOption : 0x01 TxOptACK 0x04 TxOptAutoRoute 0x20 TxOptExplore 0 = route1 0 = route2 0 = route3 0 = route4 0x08 = callbackID </pre>	<pre> [27 Jul 20 09:44:16.356] < 01 11 00 a8 00 03 46 08 31 05 05 64 00 00 a8 94 00 7e 1c 0x01 SOF 17 = size 0x00 REQ SerialCommand : 0xa8 AppCmdBridge 0x00 RxStatus - Single 3 = destnode 70 = srcnode 8 = size CommandClass : 0x31 CC_MultiSensor 0x05 Report 0x05 Sensor - Humidity 3 = precision 0 = scale 4 = size 43156 = value 0 = size 0x7e SignalStrength - Saturated 0x1c = checksum </pre>
---	---

zpiffer trace of GET (left/blue) and REPORT (right/red) for humidity reading. Value is in milli-percent.

[27 Jul 20 09:41:39.302] > 01 10 00 a9 02 46 04 31 04 01 00 25 00 00 00 00 07 10

0x01 SOF
16 = size
0x00 REQ
SerialCommand :
0xa9 SendDataBridge
2 = srcnode
70 = destnode
4 = size
CommandClass :
0x31 CC_MultiSensor
0x04 Get
0x01 Sensor - Temp
0 = scale
TxOption :
0x01 TxOptACK
0x04 TxOptAutoRoute
0x20 TxOptExplore
0 = route1
0 = route2
0 = route3
0 = route4
0x07 = callbackID

[27 Jul 20 09:41:39.332] < 01 11 00 a8 00 02 46 08 31 05 01 64 00 00 73 3d 00 7e 6b

0x01 SOF
17 = size
0x00 REQ
SerialCommand :
0xa8 AppCmdBridge
0x00 RxStatus - Single
2 = destnode
70 = srcnode
8 = size
CommandClass :
0x31 CC_MultiSensor
0x05 Report
0x01 Sensor - Temp
3 = precision
0 = scale
4 = size
29501 = value
0 = size
0x7e SignalStrength - Saturated
0x6b = checksum

zpiffer trace of GET (left/blue) and REPORT (right/red) for temperature reading. Value is in millidegrees.

[31 Jul 20 06:17:11.284] < 01 11 00 a8 00 00 00 00 7e fc

0x01 SOF
17 = size
0x00 REQ
SerialCommand :
0xa8 AppCmdBridge
0x00 RxStatus - Single
1 = destnode
80 = srcnode
8 = size
CommandClass :
0x31 CC_MultiSensor
0x05 Report
0x05 Sensor - Humidity
3 = precision
0 = scale
4 = size
9709 = value
0 = size
0x7e SignalStrength - Saturated
0xfc = checksum

[31 Jul 20 06:15:11.811] < 01 11 00 a8 00 00 00 00 7e a8

0x01 SOF
17 = size
0x00 REQ
SerialCommand :
0xa8 AppCmdBridge
0x00 RxStatus - Single
1 = destnode
80 = srcnode
8 = size
CommandClass :
0x31 CC_MultiSensor
0x05 Report
0x01 Sensor - Temp
3 = precision
0 = scale
4 = size
20439 = value
0 = size
0x7e SignalStrength - Saturated
0xa8 = checksum

Humidity (left) and temperature (right) readings after button presses.

Step 3d: ZWave - CC_CONFIGURATION

The code for this step is found [here](#).

Our data logger has run with a fixed number of samples at a fixed time interval. We want to be able to configure these by ZWave, and also to start the log remotely. We will use the CC_CONFIGURATION commands, version 1, to do this. Parameter 1 will be time in seconds between samples, Parameter 2 the number of samples, and Parameter 3 a start/stop flag.

As with the sensor command, we create new files ZAF_CC/CC_Configuration.[ch] with the header and source. There is one framework function for incoming frames, handleCCConfiguration(), which operates much like the multilevel sensor handler. It uses four functions left as prototypes for the application code to implement.

[appConfigValidParam\(\)](#) tests if the requested parameter ID is supported, [appConfigValue\(\)](#) returns a parameter's value, [appConfigLevel\(\)](#) returns the size byte for the REPORT, and [appConfigSet\(\)](#) changes a parameter, perhaps resetting it to its default value. Note that the command class defines default behavior if a bad parameter identifier or value is received, and this must be handled in the application code. The command class handler is a switch for GET, SET, and REPORT (which is ignored). The GET builds and sends a REPORT, the SET changes the logging setup and may start or stop the logger. Like the multi-byte values we had to assemble for sensor readings, with different frame structures for different sizes, here we need to unpack a SET value depending on the size indicated in the command, or fill in the REPORT with the correct number of value bytes.

We have a few changes in the application code besides implementing the four functions for the command class and adding the command class. The most important is wrapping the log in a mutex, since the timer/logging and incoming parameter changes can happen asynchronously. FreeRTOS semaphores provide the functionality and we add some limited error checking if a lock or unlock fails. You can see this in the `appConfigSet()` code snippet above. We modify the graphing functions to calculate point coordinates locally so we only have to take the lock once if we end up re-scaling the y axis. We also add a running flag to the log for when we need to check the status, because there's no way to read back the LED2 state, which would otherwise signal if logging is underway.

Besides the two command class files and ignoring edits within functions just to add the mutex lock, our application changes are

object	status	role	description
htlog_mutex, _lock (globals)	new	logger	protect asynch access to log
appnif1 (global var)	edit	transport	add command class
bound_nsampl	new	CC support	validation of nsample parameter
graph_humidity	edit	LCD driver	store points locally before drawing
graph_temperature	edit	LCD driver	store points locally before drawing
appConfigValidParam	new	CC support	validate parameter identifier
appConfigValue	new	CC support	get current parameter value
appConfigLevel	new	CC support	get size of parameter value
appConfigSet	new	CC support	change parameter value
ApplicationInit	edit	ZAF	create mutex lock for log
Transport_AppCmdHandlerEx	edit	ZAF	add command class

The project file list includes the new command class.

hw/	src/	ZAF_AppUtil/	ZAF_CC/
displayconfigapp.h displayfont16x20HT.h em_types.h graphdisplay.c graphdisplay.h textdisplay.c testdisplayconfig.h textdisplay.h	config_app.h config_rf.h HTSensor1.c i2cspmconfig.h	ZAF_TSE.c	CC_Configuration.c CC_Configuration.h CC_MultiSensor.c CC_MultiSensor.h CC_Version.c
<i>linked within Simplicity Studio</i>			
display.c displayls013b7dh03.c displaypalemlib.c em_i2c.c em_ltimer.c em_prs.c gpinterrupts.c i2cspm.c startup_zgm13.S system_zgm13.c		application_properties.c AppTimer.c board_BRD420x.c board_indicator.c board.c ZAF_command_class_utils.c zaf_event_helper.c ZAF_network_learn.c ZAF_uart_utils.c ZW_TransportEndpoint.c ZW_TransportMulticast.c ZW_TransportSecProtocol.c	CC_Supervision.c multichannel.c
<i>linked on file system</i>			
			CC_Common.h CC_Supervision.h CC_Version.h multichannel.h

Our last edits will be to add [S0 encryption](#) to the new command classes.

Step 3e: ZWave - Security

The code for this step is found [here](#).

One final change to our demo. Let's make the sensor and configuration classes secure. We'll use S0 so the zpiffer/zniiffer can decrypt the packets, although setting up S2 would be simple.

The first edit is to src/config_app.h. Define REQUESTED_SECURITY_KEYS as SECURITY_KEY_S0_BIT. For S2 you would also OR in SECURITY_KEY_S2_UNAUTHENTICATED_BIT and SECURITY_KEY_S2_AUTHENTICATED_BIT.

The second edit is to our application. We don't need to change any functions, just create two arrays with the secured command class lists and add them to the NIF. The [global arrays](#) are named cc_open_secure and cc_secure. Move the COMMAND_CLASS_MULTISENSOR_ and COMMAND_CLASS_CONFIGURATION

byte from cc_unsecure to cc_secure, and put COMMAND_CLASS_SECURITY in all three. Replace the NULL, 0 entries in appnif1 with the arrays and their length.

The third edit is to the sensor command class. Security when sending packets is controlled by flags in the transmit options. Our implementation in Step 3c has two triggers for sending a value: an incoming GET, or a button press. For the first the receive options will contain the security flags and the translation in the framework's RxToTxOptions() function will preserve them, so the reply will be encrypted if needed. For the second we faked receive options. Now, [sendSensorValue\(\)](#) will check if we have an S0 key and set the appropriate flag, before translating to transmit options and sending.

In the project

1. Link \${SDK}/ZAF/CommandClasses/Security/CC_Security.c under ZAF_CC/.
2. Link on the file system \${SDK}/ZAF/CommandClasses/Security/CC_Security.h under ZAF_CC/.

Our changes are

object	status	role	description
appnif1 (global var)	edit	transport	mark secure command classes
REQUEST_SECURITY_KEYS (config_app.h)	edit	security	enable S0
sendSensorValue (CC_MultiSensor.c)	edit	security	button press sends values securely

The final project file listing is

hw/	src/	ZAF_AppUtil/	ZAF_CC/
displayconfigapp.h displayfont16x20HT.h em_types.h graphdisplay.c graphdisplay.h textdisplay.c testdisplayconfig.h textdisplay.h	config_app.h config_rf.h HTSensor1.c i2cspmconfig.h	ZAF_TSE.c	CC_Configuration.c CC_Configuration.h CC_MultiSensor.c CC_MultiSensor.h CC_Version.c
<i>linked within Simplicity Studio</i>			
display.c displayls013b7dh03.c displaypalemlib.c em_i2c.c em_letimer.c em_prs.c gpinterrupts.c i2cspm.c startup_zgm13.S system_zgm13.c		application_properties.c AppTimer.c board_BRD420x.c board_indicator.c board.c ZAF_command_class_utils.c zaf_event_helper.c ZAF_network_learn.c ZAF_uart_utils.c ZW_TransportEndpoint.c ZW_TransportMulticast.c ZW_TransportSecProtocol.c	CC_Supervision.c CC_Security.c multichannel.c
<i>linked on file system</i>			
			CC_Common.h CC_Security.h CC_Supervision.h CC_Version.h multichannel.h

With these changes the sensor readings are now sent secured.

Inclusion done

Device inclusion is done

```
Class : COMMAND_CLASS_NETWORK_MANAGEMENT_INCLUSION
Command : NODE_ADD_STATUS
Seq. No : 00
Status : 06
Reserved : 00
New Node ID : 4e
Node Info Length : 0e
Listening : {u'Capability': '01', u'Listening': True}
Opt : {u'Opt': True, u'Security': '01'}
Basic Device Class : 04
Generic Device Class : 21
Specific Device Class : 01
Non-Secure Command Class : 986823f100317098800000
Package truncated : True
```

*zipgateway information after secured inclusion.
Command class list includes both unsecured (0x98
Security, 0x68 ZIP Name, 0x23 ZIP, 0xF1
S0_Mark) and secured (0x31 MultilevelSensor,
0x70 Configuration, 0x98 Security).*

<pre>[27 Jul 20 11:08:53.521] < 01 0c 00 4a 03 03 4e 05 04 21 01 0x01 SOF 12 = size 0x00 REQ SerialCommand : 0x4a AddNode 0x03 = callbackID 0x03 AddStatus - AddSlave 78 = nodeID 5 = count 0x04 BasicType - RoutingSlave 0x2101 Generic/SpecificType - MultiSensor/Routing 0x86 CommandClass - CC_Version 0x98 CommandClass - CC_Security 0xc8 = checksum</pre>	<pre>SerialCommand : 0xa8 AppCmdBridge 0x00 RxStatus - Single 1 = destnode 78 = srcnode 22 = size CommandClass : 0x98 CC_Security 0x81 Encap [0x 09 2d 9d 60 f3 19 2b c6] = initvec SecureMsg : 0 = second_frame 0 = sequenced 0 = seqno CommandClass : 0x98 CC_Security 0x07 VerifyNetKey 0x16 = rxnonce [0x 1f c9 43 8c f7 de cb 3f] = authcode 0 = size 0x7e SignalStrength - Saturated 0x82 = checksum</pre>
---	--

*zpiffer trace of inclusion. AppUpdate in left (blue) window shows only unsecured command classes.
Right (red) window has network key verification from device to host.*

0x00	REQ	78	= srcnode
	SerialCommand :	28	= size
0xa9	SendDataBridge		CommandClass :
3	= srcnode	0x98	CC_Security
78	= destnode	0x81	Encap
24	= size		[0x 31 89 48 bb 53 23 25 82]
	CommandClass :		= initvec
0x98	CC_Security		SecureMsg :
0x81	Encap	0	= second_frame
	[0x 9a c6 df 9c 60 41 73 82]	0	= sequenced
	= initvec	0	= seqno
	SecureMsg :		CommandClass :
0	= second_frame	0x31	CC_MultiSensor
0	= sequenced	0x05	Report
0	= seqno	0x01	Sensor - Temp
	CommandClass :	3	= precision
0x31	CC_MultiSensor	0	= scale
0x04	Get	4	= size
0x01	Sensor - Temp	29630	= value
0	= scale	0x9b	= rxnonce
0x44	= rxnonce		[0x fa 1e ec f2 4d 1f 44 bc]
	[0x 84 ee 1a 90 fe 9c ff 71]		= authcode
	= authcode	0	= size
	TxOption :	0x7e	SignalStrength - Saturated

Secured GET (left/blue) and REPORT (right/red) sequence for temperature reading. Nonce frame not shown.

78	= srcnode
28	= size
	CommandClass :
0x98	CC_Security
0x81	Encap
	[0x fb b7 6c 44 7b 6f d1 b8]
	= initvec
	SecureMsg :
0	= second_frame
0	= sequenced
0	= seqno
	CommandClass :
0x31	CC_MultiSensor
0x05	Report
0x05	Sensor - Humidity
3	= precision
0	= scale
4	= size
39532	= value
0x21	= rxnonce
	[0x ee 92 f1 a2 f6 45 57 e7]
	= authcode
0	= size
0x7e	SignalStrength - Saturated

Button press sends secured humidity reading.

We'll [wrap up](#) the project with a review of all that's been done and an introduction to the other pieces of the framework.

Step 4: Application Framework

This brings us to the end of the demo. We've learned how projects are set up in Simplicity Studio, created one from scratch, and added in the SDK hardware and ZAF files needed to compile a program within the framework. We needed to link some hardware source files, used some code libraries (components), added libraries for Zwave and radio testing, defined a few symbols for changing the configuration of the hardware files, and set many many directories on the include path. The Application Framework runs under FreeRTOS and, at the the application level at least, uses a single task and four event handlers. Events can come from the hardware (button presses), state changes within the framework (EZWCOMMANDSTATUS_*), or incoming frames routed by the packet type (EZWAVERECEIVETYPE_*) or, for ZWave commands embedded in a frame, by the command

class (ZW_Common member of ZW_APPLICATION_TX_BUFFER). The packet type router uses the CommandPublisher to unpack a frame and send it to the command handler, if the transport layer hasn't already done so. New ZWave command classes are added in a file with a function that acts on the specific command (ZW_Command member of ZW_APPLICATION_TX_BUFFER); it may also have an outgoing path for application events (for us, button presses) and replies. Security is handled automatically according to the contents of the node information frame (NIF).

We can define the skeleton of a ZAF project; it will include a minimum amount of configuration information, a FreeRTOS task, event handlers, and whichever ZWave commands are needed by the product.

1. Define version, device type, manufacturer and product ID, radio parameters, number of endpoints, and security in config_app.h and config_rf.h.
2. Define the NIF contents in a global variable, holding arrays of command classes grouped by security level.
3. Implement ApplicationInit(), which does initial set up and registers the task with the application.
 - A. Provide ApplicationTask() which does board level set up and creates the event queue, then enters the main loop that distributes events to one of four handlers:
 - i. AppTimerNotificationHandler(), defined by ZAF.
 - ii. handle_frame(), which routes incoming frames based on received packet type, and which calls the CommandPublisher to unpack embedded ZWave commands.
 - iii. handle_cmdstatus(), which routes framework/ZWave state changes, for example for learning.
 - iv. handle_event(), which routes hardware events.
4. Implement Transport_ApplicationCommandHandlerEx(), which sends frames that have been unpacked by the transport layer to the appropriate command class.
5. Implement application-specific functionality for the command classes. This depends on the command class; see binary switch and user code for example, or for this project, the multilevel sensor and configuration commands.

The rest of the demo implements functionality triggered off these events or command classes.

The Framework contains more than 60 source and header files, excluding the command classes. We've only used a small part of what it offers. Unfortunately the only documentation seems to be reading the source. As a central reference, we've used the following functions:

AppTimer.[ch]

function	role	called from
AppTimerInit	set up	ApplicationInit()
AppTimerSetReceiverTask	cache FreeRTOS task in timer	ApplicationTask()
AppTimerRegister	set callback function when timer expires	ApplicationTask()
AppTimerNotificationHandler	framework handler for timer events	ApplicationTask()

board.[ch]		
function	role	called from
BTN_EVENT_*	macros to decode button presses	HW event handler
Board_Init	set up	ApplicationInit()
Board_SetLed	turn LED on or off	internal
Board_IndicatorInit	set up LED for learn mode	ApplicationTask()
Board_IndicatorControl	turn indicator on/off or set flashing	internal
Board_EnableButton	set up push buttons	ApplicationTask()
ZAF_CmdPublisher.[ch]		
function	role	called from
ZAF_CP_CommandPublish	route singlecast packet to commands	frame handler
ZAF_network_learn.[ch]		
function	role	called from
ZAF_setNetworkLearnMode	start/stop learn mode from push button	HW event handler
ZAF_uart_utils.[ch]		
function	role	called from
ZAF_UART0_enable	start UART for debug logging (DPRINTF)	ApplicationInit()
ZAF_UART0_tx_send	send debug log over UART	ApplicationInit()
ZW_TransportEndpoint.[ch]		
function	role	called from
Transport_SendResponseEP	transmit REPORT or other response	command class
Transport_Application	handle incoming frames by packet type	event distributor
CommandHandlerEx		
RxToTxOptions	convert receive serial options to transmit	command class
Check_not_legal_response_job	verify multicast/supervision encapsulation	command class
ZAF_Common_interface.h		
function	role	called from
ZAF_getAppHandle	get pointer to ZWave configuration, queues	HW event handler
ZAF_GetSecurityKeys	bit-encoded list of security keys available	command class
ZAF_getCPHandle	get pointer to CommandPublisher	frame handler
zaf_event_helper.[ch]		
function	role	called from
ZAF_EventHelperInit	start event queue	ApplicationTask
ZAF_TSE.[ch]		
function	role	called from
ZAF_TSE_TXCallback	callback for Transport_SendRequestEP	ZAF status handler

The Called From column indicates where the function is used in the demo. It isn't clear what has to go in ApplicationInit() and what in ApplicationTask().

A complete ZWave device must implement more functionality. ZAF is designed to easily support these extra requirements, including all command classes for ZWave+, Smart Start, and application groups and the True Search Engine (lifeline support). The five demo programs in the SDK have a similar architecture to each other. They have more application states than ours, and use the event queue not just for ZWave functionality but also

for custom events that manage the application state. The logic defining the state machine/event handler is often fairly involved. Comparing our demo against those in the SDK, as well as the ZAF specification INS14259, gives us an idea of the additional work we would need to do to make this a full-fledged product.

ZWave+ Command Classes. At the start of Step 3a we listed the command classes required by ZWave+. The common ones are available in the SDK under ZAF/CommandClasses and can be included as a project module or by linking the source and header files into the project, as we did for CC_Security and CC_Supervision.

Application Events. We have used only the events ZAF generates for button presses in `handle_event()`. The SDK demos use a richer set to handle basic functionality as well as application-specific states, and they use a separate job queue to temporarily store work when reacting to button presses in the event loop. The events are defined in the header file `events.h` in the source directory; ZAF provides no standard list, and all are application-specific. The complete list over the five SDK demos shows us the type of functionality they track; a star (*) before an event means it's provided in most of the demos and can be considered key functionality.

EVENT_APP_*		Lock	Power Strip	Sensor PIR	OnOff Switch	Wall Ctlr
*	INIT		X	X	X	X
*	REFRESH_MMI	X	X		X	X
*	FLUSHMEM_READY	X	X	X	X	X
*	SMARTSTART_IN_PROGRESS	X	X	X	X	X
*	LEARN_IN_PROGRESS	X	X	X	X	X
	LEARN_MODE_FINISH					X
	TOGGLE_LEARN_MODE					X
*	NEXT_EVENT_JOB	X		X		X
*	FINISH_EVENT_JOB	X	X	X		X
	BASIC_STOP_JOB			X		
	BASIC_START_JOB			X		
	NOTIFICATION_START_JOB			X		
	NOTIFICATION_STOP_JOB			X		
	START_TIMER_EVENTJOB_STOP			X		
	CC_BASIC_JOB					X
	CC_SWITCH_MULTILEVEL_JOB					X
	CENTRAL_SCENE_JOB					X
	SEND_OVERLOAD_NOTIFICATION		X			
	SEND_BATTERY_LEVEL_REPORT	X		X		
	IS_POWERING_DOWN	X		X		
	START_USER_CODE_EVENT	X				
	START_KEYPAD_ACTIVE	X				
	FINISH_KEYPAD_ACTIVE	X				
	PERIODIC_BATTERY_CHECK_TRIGGER	X				

REFRESH_MMI handles the network indicator. FLUSHMEM_READY resets non-volatile memory and reloads the application configuration or does a soft reset, depending on the application state. SMARTSTART_IN_PROGRESS, LEARN_IN_PROGRESS, LEARN_MODE_FINISH, and TOGGLE_LEARN_MODE track progress through inclusion and transition into SmartStart modes.

Job Queue. The job queue is used to delay executing actions while the application is busy processing something else. It stores application events. The demos push NEXT_EVENT_JOB on the event queue which, when handled, will take the top job event and push it onto the event queue, or FINISH_EVENT_JOB if there are no jobs. Only one event is moved at a time, to disturb the main event flow as little as possible. NEXT_EVENT_JOB is typically pushed after a transmission is done, either if it's sent directly or in a callback. The use of the job queue can be quite simple, as in the PowerStrip demo where it's used to cleanup a notification

and switch to the idle application state, or complex, as in the WallController demo which uses it to work through central scene notifications, or may not be used at all, as in the SwitchOnOff demo.

SmartStart. The SDK demos implement SmartStart by finer control of the application state. They define an enumeration `_STATE_APP_` to track progress through start-up and inclusion, and a function `AppStateManager()` to transition through the states as application events come in. In our demo we've simplified the state to a couple of booleans, tracking if the log is running or we're including or excluding the device, and transitions are scattered through the code. The `AppStateManager()` replaces our `handle_event()`. SmartStart learn mode begins in `EVENT_APP_INIT`, the initialization state, with a call to `ZAF_setNetworkLearnMode()`. It also triggers when learning is stopped by a button press (`Button1` in our list), if inclusion fails, or if the device is excluded from a network.

Association Groups. Association groups are created with several defines in `app_config.h`, including `NUMBER_OF_ENDPOINTS`, `MAX_ASSOCIATION_GROUPS`, `MAX_ASSOCIATION_IN_GROUP`, `AGITABLE_LIFELINE_GROUP`, `AGITABLE_ROOTDEVICE_GROUPS`, `AGITABLE_ENDPOINT*_GROUPS`, and `ASSOCIATION_ROOT_GROUP_MAPPING_CONFIG`. The first three are numbers, the fourth a comma-separated list of pairs of command classes and commands, and the remaining are described in INS14259 Section 5.2.4. The application contains globals to hold the setup, with `agiTableLifeLine[]` initialized to `AGITABLE_LIFELINEGROUP` and `agiTableRootDeviceGroups[]` to `AGITABLE_ROOTDEVICE_GROUPS`. These are passed to `CC_AGI_LifeLineGroupSetup()` and `AGI_ResourceGroupSetup()` during application initialization. The WallController demo has a simple association group setup, the PowerStrip a much more complicated one, with multiple endpoints. Sections 5.6.2 and 7.1.1 of INS14259 also have more details.

True Status Engine. The True Status Engine is used to send reports and notifications to the lifeline after a short delay to avoid packet collisions and spamming the network. Command classes will define callback functions `CC_<cmdclass>_report_stx()`. The function has two arguments, transmit options with the destination filled in, and a command-specific data structure that includes as its first member a `RECEIVE_OPTIONS_TYPE_EX` structure with information about the incoming frame. The callback will build the appropriate frame and will call `Transport_SendRequestEP()`. The application should call `ZAF_TSE_Trigger()` on an event or state change that generates traffic to the lifeline.

NVM. To use non-volatile memory the application must define a structure to hold application data that must persist over power loss, and an associated global variable. Accessing memory is done through the driver in `${SDK}/platform/emdrv/nvm3` and `${SDK}/ZAF/ApplicationUtilities/ZAF_nvm3_app.[ch]`. Call `ApplicationFileSystemInit()` to get a pointer to the file system stored in NVM, normally when loading the configuration during application initialization. `${SDK}/ZAF/ApplicationUtilities/ZAF_file_ids.h` defines some standard entries that the framework or command classes will use or provide:

<code>ZAF_FILE_ID_APP_VERSION</code>	ZAF version numbers (required)
<code>ZAF_FILE_ID_ASSOCIATIONINFO</code>	groups per endpoint
<code>ZAF_FILE_ID_USERCODE</code>	receive options, user ID
<code>ZAF_FILE_ID_BATTERYDATA</code>	last reported level
<code>ZAF_FILE_ID_NOTIFICATIONDATA</code>	alarm status
<code>ZAF_FILE_ID_WAKEUPCCDATA</code>	master node and sleep period

Each has an accompanying ZAF_FILE_SIZE_* and both are needed in the global variable that contains all file system entries. Application-specific data is defined by setting a file identifier, normally 0, and the size of the structure. The first can go in config_app.h, the second in the application source. For example, a binary switch might need to store the application version, association groups, battery level, and switch state. The local NVM structure and combined file system would look like:

```
struct NVMApplData { uint8_t switchOn; }
SFileDescriptor fileIDs[] = {
    { .ObjectKey=ZAF_FILE_ID_APP_VERSION, .iDataSize=ZAF_FILE_SIZE_APP_VERSION },
    { .ObjectKey=ZAF_FILE_ID_ASSOCIATIONINFO, .iDataSize=ZAF_FILE_SIZE_ASSOCIATIONINFO },
    { .ObjectKey=ZAF_FILE_ID_BATTERYDATA, .iDataSize=ZAF_FILE_SIZE_BATTERYDATA },
    { .ObjectKey=0x00, .iDataSize=sizeof(struct NVMApplData) }
}
EFileSysVerifyStatus fileStatus[sizeof_array(fileIDs)];
```

Associations are stored after learning is complete with a call to AssociationInit(). The application version is stored when initializing the NVM file system or after a reset with a call to nvm3_writeData(). Command classes may or may not handle their own file: wake-up and notification do, battery and user code do not. If not then the application is responsible for saving changes. Getting information out of NVM is done with nvm3_readData(). Both take the file ID, a pointer to the source or destination memory, and the size of the structure. Application data should be written after every state change.

Power Management. There are five power management modes: EM0 (active), EM1 (sleep), EM2 (deep sleep), EM3 (stop), EM4 (hibernate), and EM4 (shutdown). The data sheet DSH14299 defines which hardware modules are active in which modes. The LETimer, for example, is available in EM0, EM1, and EM2, as is the LESense sensor interface. Peripherals are grouped in two domains. The first includes one analog comparator, the pulse counter, ADC, LETimer, LESense, and analog port. The second has the other analog comparator, capsense, voltage and current DACs, UART, and I2C. If all members of a group are unused then all are powered off in low-power modes. Wake-up times vary, being 3 clocks out of EM1, 4-11 us from EM2 or EM3, 90 us from hibernate, and 300 us from shutdown. The power manager is set up in `SDK/ZWave/API/ZW_PowerManager_api.h`, which defines a power level `SPowerLock_t` to prevent the chip from entering a lower power mode, and `SDK/ZAF/ApplicationUtilities/PowerManagement/ZAF_PM_Wrapper.h`, which has functions to set this level. `PM_TYPE_RADIO` will prevent the chip from entering EM2-4, `PM_TYPE_PERIPHERAL` from EM3-4. `ZAF_PM_StayAwake()` and `ZAF_PM_Cancel()` bracket the code section where the power level is set. `ZAF_PM_Register()` sets up the lock for the power level. For example, the wakeup command class will set `PM_TYPE_RADIO` for 10 seconds after a wake-up notification, canceling it early after a No More Information has been sent. Note that the AppTimer also provides EM4 functionality.